ACM SIGGRAPH 2001 Course #16

Gaming Techniques for Designing Compelling Virtual Worlds

Course Organizer:

Michael Capps
Naval Postgraduate School

For updated course notes, links to demonstrations, etc., see:
http://sharedvr.org/learn

Instructors:

Yahn Bernier, Valve, LLC
Cliff Bleszinski, Epic Games
Michael Capps, Naval Postgraduate School
Shane Caudle, Epic Games
Jesse Schell, Walt Disney Imagineering VR Studio

Abstract:

This course presents the world-building tricks of the computer game trade, which is a multi-billion dollar competition to build the most enticing and immersive virtual environments. Speakers describe their approaches to designing environments, review their experiences (both good and bad), and showcase their latest technologies.

The topics presented will include issues of community building and society design; world navigability by humans and artificially intelligent software agents; people flow and bottlenecks; and the impact of design on technology, and technology on design.

These notes contain:

- Course outline
- Speaker Biographies
- "Creating Immersive Multiplayer Action Experiences" [slides]
- "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization" [paper]
- "Art and Science of Level Design" [paper]
- "Panda3D" [paper]
- Three chapters excerpted from upcoming *3D Game Art f/x and Design* by Luke Ahearn

Gaming Techniques for Designing Compelling Virtual Worlds

Time	**Speaker**	**Topic**
1:30	Capps	**Introduction**
		Building a Virtual Reality from Reality – *Worlds with realistic terrain* – *Building 3D models from real objects* – *Case study and demonstration: the Army Game Project*
2:00	Bleszinksi and Caudle	**Making Compelling Worlds** – *Looks and functionality* – *Flow* – *Scene composition* – *Dramatic lighting and sound* – *Level geometry* – *Hardware brushes and prefabricated objects* – *Texturing detail vs. geometric detail* – *Case study and demonstration: Epic Games' Unreal Engine*
2:50	Bernier	**Creating Immersive Multiplayer Action Experiences** – *Maintaining immersion* – *World and control responsiveness* – *Client/Server architecture* – *Prediction and extrapolation* – *Lag compensation* – *Networking paradoxes* – *Case study and demonstration: Valve Software's Half-Life and TeamFortress*
3:40	Schell	**Designing for Community** – *Lessons from theme park design* – *Geography and motion* – *Social interaction as a driver for environment geography* – *Clustering* – *Case study and demonstration: Disney's Toon Town Online*
4:30	All	**Conclusion and Q/A**

ORGANIZER BIOGRAPHY

Michael Capps
Research Assistant Professor
Naval Postgraduate School
Code CS/Cm, Dept of Computer Science
Monterey, CA 93943-5118
Email address: capps@acm.org

Michael Capps is a professor in the Modeling, Virtual Environments, and Simulation curriculum at the Naval Postgraduate School. His research involves techniques for optimization of networked graphics, and software engineering for interoperable shared virtual environments. Of late, he has focused on applying virtual environment research results to multiplayer entertainment software, and consults actively in that area. Other interests include geometric reconstruction, collaborative computing, and hypertext protocols; he has published technical academic articles ranging across all of these topics. Michael organizes the annual "Systems Aspects of Sharing a Virtual Reality" workshop series, which is in its fourth year. He received honors in Mathematics and Creative Writing in his undergraduate work at the University of North Carolina; he holds graduate degrees in Computer Science from the University of North Carolina, MIT, and the Naval Postgraduate School. Michael has served on the organizing committee of several prominent academic conferences, including ACM Hypertext, CSCW, VRAIS, IEEE Virtual Reality, and the annual World Wide Web conference series; he was Technical Program Chair for the 2000 VRML Symposium and the 2001 Web3D Symposium.

Dr. Capps' primary responsibilities now lie with the Army Game Project, a multiple year effort to apply off-the-shelf gaming technology to military simulation and training efforts. More information on the project can be found at www.armygame.com.

INSTRUCTOR BIOGRAPHIES

Yahn Bernier
Sr. Software Development Engineer
Valve, LLC
520 Kirkland Way, Suite 200
Kirkland, WA 98033
email: yahn@valvesoftware.com

Yahn received his undergraduate degree in Chemistry from Harvard University. He then went on to study law at the University of Florida School of Law. After law school, Yahn moved to Atlanta and spent five years practicing patent law there. Yahn's law practice was focused in the areas of computer software, chemistry, biochemistry, and mechanical engineering. In his spare time, he authored the popular "Quake" level editor BSP and because of this work he was contacted by Valve, LLC in late 1997. After receiving the proverbial offer that was "too good to refuse," he moved to Seattle where he began working on technology for Valve's first title: Half-Life. Currently, Yahn is responsible for the network aspects of Valve's future titles, including TeamFortress 2 on which he is the technical lead. His work includes not only the in-game data flow, but also the various external components and services that comprise Valve's gaming platform.

Cliff Bleszinski
Lead Designer, Epic Games, Inc.
5511 Capital Center Drive, Suite 675
Raleigh, NC 27606
Email address: cliff@epicgames.com

Cliff began his game career at age 17, with the 1993 release of Epic's Jazz Jackrabbit. Cliff was later the Lead Designer for Epic's Unreal, which brought new levels of world design to the first-person perspective game genre. He also was Lead Designer for the latest Unreal product, Unreal Tournament, which has sold over 1,000,000 units and was named Game of the Year in multiple major gaming publications.

Shane Caudle
Art Director, Epic Games, Inc.
5511 Capital Center Drive, Suite 675
Raleigh, NC 27606
Email address: shane@scott.net

Shane Caudle currently works for Epic Games as Art Director, where he provides artwork, 3D models, animations, game textures, and level designs. He worked for Epic on both Unreal and Unreal Tournament. Prior to that, Shane was an animator and artist for a company called Rival Productions, which he founded. At Rival, Shane worked on a 2D/3D comic book called "Eye of the Storm," along with a variety of animation for TV, computer games, and movie pilots.

Jesse Schell
Game Designer / Programmer
Walt Disney Imagineering VR Studio
1401 Flower St.
Glendale, CA 91221
Email address: jesse.schell@wdi.disney.com

Jesse Schell is a show programmer and game designer at the Walt Disney Imagineering Virtual Reality Studio. He has helped develop such attractions as:

- "Aladdin's Magic Carpet Ride", a head mounted display based VR attraction, currently installed at DisneyQuest (Disney's chain of interactive indoor theme parks);
- "Hercules in the Underworld", a CAVE-based VR attraction, also at DisneyQuest;
- "Mickey's Toontown Tag", a multiplayer game currently installed at Epcot Center at Disneyworld; and
- An interactive "Pirates of the Caribbean" multiplayer CAVE-based VR attraction, soon to open at DisneyQuest.

Pre-Disney work includes:

- Designing artificial intelligence systems for automated storytelling as part of Rensselaer's Autopoeisis project.
- Designing and building the NVR system for networked virtual reality at Carnegie Mellon. This system was an integral part of the "Virtual Reality: An Emerging Medium" exhibit at the Guggenheim Soho in New York City.
- Co-writing, directing, and hosting the comedy radio show, "Laughter Hours"
- Writing, directing, and performing as a juggler, comedian, and circus artist in shows with both the Juggler's Guild and Freihofer's Mime Circus entertainment troupes.

Jesse has a B.S. in Computer Science from Rensselaer, and an M.S. in Information Networking from Carnegie Mellon. His main interest is making virtual worlds more *fun*.

Creating Immersive Multiplayer Action Experiences

Yahn W. Bernier
Senior Software Development Engineer
Valve
520 Kirkland Way, Suite 200
Kirkland, WA 98033
425-889-9642
yahn@valvesoftware.com

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Overview

Immersion

Control response

World response

Network effects

- Heterogonous Network Environments
- Transient Network Reliability Issues

Control Response

Local controls should respond with minimal latency

Framerate *is* an issue

Players can detect control sampling rates

Network latency hinders responsiveness

Consider automating common user tasks

World Response

Player can manipulate most objects

Player can mark up the world (decals)

Sounds from objects should be spatialized correctly & DSP effects should be employed

Robust animation (no ice-skating) implies player intentionality

Network Effects

- Latency hinders responsiveness
- Control inputs must be predicted on client
- Local weapon actions should be predicted on client
- Consider letting server give credit for local weapon actions (lag compensation)

Things Which Hinder Immersion

- Latency of user action or response to action
- User actions with no visible consequences
- Inconsistency in the way objects react
- Time synchronization paradoxes and prediction errors
- Non-believable avatar movements/actions

Half-Life (*et al*.) Client / Server Game Architecture

Authoritative Server versus Peer-to-Peer

- Cheating is major issue and quickly destroys communities

Simple client

User Input / Control processing

World state / User Input results

The frame loop

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

User Input / Control Input

Encapsulation of control inputs (keyboard, mouse, joystick) allows for simulation to run as a component

Basic control inputs

- Time slice
- Movement/view direction
- Movement impulses & other action/button states

Client Frame Loop

Check controls and use Simulation time to encapsulate User Input

Send User Input to Server

Read server packets

Render visible objects from server packets

Compute next Simulation Time

Server Frame Loop

Read client packets

Process User Input in packets

Simulate other server objects using server Simulation Time

Send clients world state update

Compute next Simulation Time

- Note that client's drive their own Simulation Time clock (server bounds clock to prevent stragglers)

Client–Side Movement Prediction

- Most world state is coherent
- Client has sufficient info to “guess” at User Input results
- Discards simple client model, but server remains authoritative
- Result - user control inputs are highly responsive and the player is immersed in the game

Client–Side Movement Prediction

- Most recently received player state – User Input last acknowledged by the server
- For each User Input not yet acknowledged, run player simulation locally to generate a new player state
- Repeat until the current client time

Client–Side Movement Prediction

Effects/sounds/decals only created first time a User Input is predicted

Effects/sounds/decals are not sent from the server to the predicting client (but are sent to other clients)

Solve the problem of prediction errors

Client–Side Weapons

Similar to movement – player inputs must respond crisply

Must store additional state

Must be able to run the weapon logic locally

- Shared code, common interface?
- Separate implementations on client/server?

Weapon effects, including marking up the world with smoke/bullet marks, should all occur locally

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Presentation

- Reconciling the server and client clock and dealing with network effects
- Extrapolation vs. Interpolation
- Hybrid / Continuous Model
- Immersiveness is increased by smooth presentations of other players even under high latency/high packet loss situations

Extrapolation

Use last known velocity and position

- Subtract last two player positions to determine delta
- Subtract last two player time stamps to determine elapsed time for the delta
- Compute apparent velocity

Compute new position based on x = x0 + (time) * velocity

Extrapolations should be capped to avoid severe errors under lag/packet loss

Interpolation – Method 1

Target is position in last received update

Interpolation time is subtracted from client clock

Object is moved from last render position toward target

Object reaches target when the client clock is the interpolation time ahead of the last update time stamp

Interpolation – Method 2

Keep database of the positions and timestamps for objects

Interpolation time is subtracted from the current client clock to determine target time

Search database for the two updates that span the target time

Render position is linearly interpolated between the two spanning updates

Lag Compensation

Another tool to maintain immersion

Relies on all of the preceding techniques

Interpolation adds latency that must be factored

Connection latency must also be factored

Server must reconcile predicted and interpolated client world view and determine results of player actions

Lag Compensation

Server computes accurate latency for player

Server finds best world update sent to the player

Server moves other players backward in time

Player's User Input is executed on "reconstructed" world state

Other players are returned to current time positions

Lag compensation & Game Design

- Increases immersion & world and control responsiveness
- Eliminates guesswork but not aiming skill from weapon firing
- Not as suitable for projectiles and melee/hand-to-hand interactions

Lag Compensation Paradoxes

From the attackee's POV

Shooting around corners

Head-on versus perpendicular traversal

Games with swiveling player heads

Conclusion

Make your controls responsive

Make your world responsive

Be proactive about network effects

Make sure everything behaves well for the local player, even with high latency and high packet loss

Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization

Yahn W. Bernier
Valve
520 Kirkland Way, Suite 200
Kirkland, WA 98033
(425) 889-9642
e-mail: yahn@valvesoftware.com

Overview:

Designing first-person action games for Internet play is a challenging process. Having robust on-line gameplay in your action title, however, is becoming essential to the success and longevity of the title. In addition, the PC space is well known for requiring developers to support a wide variety of customer setups. Often, customers are running on less than state-of-the-art hardware. The same holds true for their network connections.

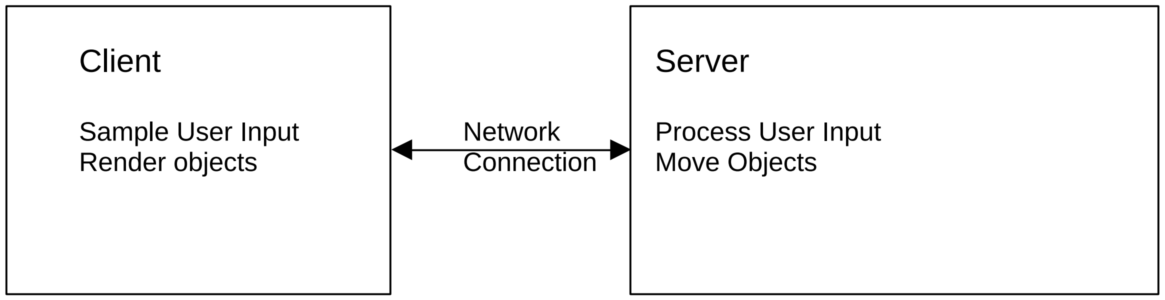
While broadband has been held out as a panacea for all of the current woes of on-line gaming, broadband is not an simple solution allowing developers to ignore the implications of latency and other network factors in game designs. It will be some time before broadband truly becomes adopted the United States, and much longer before it can be assumed to exist for your clients in the rest of the world. In addition, there are a lot of poor broadband solutions, where users may occasionally have high bandwidth, but more often than not also have significant latency and packet loss in their connections.

Your game must to behave well in this world. This discussion will give you a sense of some of the tradeoffs required to deliver a cutting-edge action experience on the Internet. The discussion will provide some background on how client / server architectures work in many on-line action games. In addition, the discussion will show how predictive modeling can be used to mask the effects of latency. Finally, the discussion will describe a specific mechanism, lag compensation, for allowing the game to compensate for connection quality.

Basic Architecture of a Client / Server Game

Most action games played on the net today are modified client / server games. Games such as Half-Life, including its mods such as Counter-Strike and Team Fortress Classic, operate on such a system, as do games based on the Quake3 engine and the Unreal Tournament engine. In these games, there is a single, authoritative server that is responsible for running the main game logic. To this are connected one or more “dumb” clients. These clients, initially, were nothing more than a way for the user input to be sampled and forwarded to the server for execution. The server would execute the input commands, move around other objects, and then send back to the client a list of objects to render. Of course, the real world system has more components to it, but the simplified breakdown is useful for thinking about prediction and lag compensation.

With this in mind, the typical client / server game engine architecture generally looks like:



Send current objects to Client for rendering

Figure 1: General Client / Server Architecture

For this discussion, all of the messaging and coordination needed to start up the connection between client and server is omitted. The client's frame loop looks something like the following:

```
Sample clock to find start time

Sample user input (mouse, keyboard, joystick)

Package up and send movement command using simulation time

Read any packets from the server from the network system

Use packets to determine visible objects and their state
Render Scene

Sample clock to find end time

End time minus start time is the simulation time for the next frame
```

Each time the client makes a full pass through this loop, the "frametime" is used for determining how much simulation is needed on the next frame. If your framerate is totally constant then frametime will be a correct measure. Otherwise, the frametimes will be incorrect, but there isn't really a solution to this (unless you could deterministically figure out exactly how long it was going to take to run the next frame loop iteration before running it…).

The server has a somewhat similar loop:

```
Sample clock to find start time

Read client user input messages from network

Execute client user input messages

Simulate server-controlled objects using simulation time from last full pass

For each connected client, package up visible objects/world state and send to
client

Sample clock to find end time

End time minus start time is the simulation time for the next frame
```

In this model, non-player objects run purely on the server, while player objects drive their movements based on incoming packets. Of course, this is not the only possible way to accomplish this task, but it does make sense.

Contents of the User Input messages

In Half-Life engine games, the user input message format is quite simple and is encapsulated in a data structure containing just a few essential fields:

```
typedef struct usercmd_s
{
      // Interpolation time on client
      short       lerp_msec;
      // Duration in ms of command
      byte        msec;
      // Command view angles.
      vec3_t      viewangles;
      // intended velocities
      // Forward velocity.
      float       forwardmove;
      // Sideways velocity.
      float       sidemove;
      // Upward velocity.
      float       upmove;
      // Attack buttons
      unsigned short  buttons;
      //
      // Additional fields omitted…
      //
} usercmd_t;
```

The critical fields here are the msec, viewangles, forward, side, and upmove, and buttons fields. The msec field corresponds to the number of milliseconds of simulation that the command corresponds to (it's the frametime). The viewangles field is a vector representing the direction the player was looking during the frame. The forward, side, and upmove fields are the impulses determined by examining the keyboard, mouse, and joystick to see if any movement keys were held down. Finally, the buttons field is just a bit field with one or more bits set for each button that is being held down.

Using the above data structures and client / server architecture, the core of the simulation is as follows. First, the client creates and sends a user command to the server. The server then executes the user command and sends updated positions of everything back to client. Finally, the client renders the scene with all of these objects. This core, though quite simple, does not react well under real world situations, where users can experience significant amounts of latency in their Internet connections. The main problem is that the client truly is "dumb" and all it does is the simple task of sampling movement inputs and waiting for the server to tell it the results. If the client has 500 milliseconds of latency in its connection to the server, then it will take 500 milliseconds for any client actions to be acknowledged by the server and for the results to be perceptible on the client. While this round trip delay may be acceptable on a Local Area Network (LAN), it is not acceptable on the Internet.

Client Side Prediction

One method for ameliorating this problem is to perform the client's movement locally and just assume, temporarily, that the server will accept and acknowledge the client commands directly. This method is can be labeled as client-side prediction.

Client-side prediction of movements requires us to let go of the "dumb" or minimal client principle. That's not to say that the client is fully in control of its simulation, as in a peer-to-peer game with no central server. There still is an authoritative server running the simulation just as noted above. Having an authoritative server means that even if the client simulates different results than the server, the server's results will eventually correct the client's incorrect simulation. Because of the latency in the connection, the correction might not occur until a full round trip's worth of time has passed. The downside is that this can cause a very perceptible shift in the player's position due to the fixing up of the prediction error that occurred in the past.

To implement client-side prediction of movement, the following general procedure is used. As before, client inputs are sampled and a user command is generated. Also as before, this user command is sent off to the server. However, each user command (and the exact time it was generated) is stored on the client. The prediction algorithm uses these stored commands.

For prediction, the last acknowledged movement from the server is used as a starting point. The acknowledgement indicates which user command was last acted upon by the server and also tells us the exact position (and other state data) of the player after that movement command was simulated on the server. The last acknowledged command will be somewhere in the past if there is any lag in the connection. For instance, if the client is running at 50 frames per second (fps) and has 100 milliseconds of latency (roundtrip), then the client will have stored up five user commands ahead of the last one acknowledged by the server. These five user commands are simulated on the client as a part of client-side prediction. Assuming full prediction[1], the client will want to start with the latest data from the server, and then run the five user commands through "similar logic" to what the server uses for simulation of client movement. Running these commands should produce an accurate final state on the client (final player position is most important) that can be used to determine from what position to render the scene during the current frame.

In Half-Life, minimizing discrepancies between client and server in the prediction logic is accomplished by sharing the identical movement code for players in both the server-side game

[1] In the Half-Life engine, it is possible to ask the client-side prediction algorithm to account for some, but not all, of the latency in performing prediction. The user could control the amount of prediction by changing the value of the "pushlatency" console variable to the engine. This variable is a negative number indicating the maximum number of milliseconds of prediction to perform. If the number is greater (in the negative) than the user's current latency, then full prediction up to the current time occurs. In this case, the user feels zero latency in his or her movements. Based upon some erroneous superstition in the community, many users insisted that setting pushlatency to minus one-half of the current average latency was the proper setting. Of course, this would still leave the player's movements lagged (often described as if you are moving around on ice skates) by half of the user's latency. All of this confusion has brought us to the conclusion that full prediction should occur all of the time and that the pushlatency variable should be removed from the Half-Life engine.

code and the client-side game code. These are the routines in the “pm_shared/” (which stands for “player movement shared”) folder of the HL SDK (http://download.cnet.com/downloads/0-10045-100-3422497.html). The input to the shared routines is encapsulated by the user command and a “from” player state. The output is the new player state after issuing the user command. The general algorithm on the client is as follows:

```
“from state” <- state after last user command acknowledged by the server;

“command” <- first command after last user command acknowledged by server;

while (true)
{
      run “command” on “from state” to generate “to state”;
      if (this was the most up to date “command”)
            break;

      “from state” = “to state”;
      “command” = next “command”;
};
```

The origin and other state info in the final “to state” is the prediction result and is used for rendering the scene that frame. The portion where the command is run is simply the portion where all of the player state data is copied into the shared data structure, the user command is processed (by executing the common code in the pm_shared routines in Half-Life’s case), and the resulting data is copied back out to the ”to state”.

There are a few important caveats to this system. First, you’ll notice that, depending upon the client’s latency and how fast the client is generating user commands (i.e., the client’s framerate), the client will most often end up running the same commands over and over again until they are finally acknowledged by the server and dropped from the list (a sliding window in Half-Life’s case) of commands yet to be acknowledged. The first consideration is how to handle any sound effects and visual effects that are created in the shared code. Because commands can be run over and over again, it’s important not to create footstep sounds, etc. multiple times as the old commands are re-run to update the predicted position. In addition, it’s important for the server not to send the client effects that are already being predicted on the client. However, the client still must re-run the old commands or else there will be no way for the server to correct any erroneous prediction by the client. The solution to this problem is easy: the client just marks those commands which have not been predicted yet on the client and only plays effects if the user command is being run for the first time on the client.

The other caveat is with respect to state data that exists solely on the client and is not part of the authoritative update data from the server. If you don’t have any of this type of data, then you can simply use the last acknowledged state from the server as a starting point, and run the prediction user commands “in-place” on that data to arrive at a final state (which includes your position for rendering). In this case, you don’t need to keep all of the intermediate results along the route for predicting from the last acknowledged state to the current time. However, if you are doing any logic totally client side (this logic could include functionality such as determining where the eye position is when you are in the process of crouching—and it’s not really totally client side since the server still simulates this data also) that affects fields that are not replicated from the server to the client by the networking layer handling the player's state info, then you will need to store the intermediate results of prediction. This can be done with a sliding window, where the "from state" is at the start and then each time you run a user command through prediction, you fill in the next state in the window. When the server finally acknowledges receiving one or more commands that had been predicted, it is a simple matter of looking up which state the server is acknowledging and copying over the data that is totally client side to the new starting or "from state".

So far, the above procedure describes how to accomplish client side prediction of movements. This system is similar to the system used in QuakeWorld[2].

Client-Side Prediction of Weapon Firing

Layering prediction of the firing effects of weapons onto the above system is straightforward. Additional state information is needed for the local player on the client, of course, including which weapons are being held, which one is active, and how much ammo each of these weapons has remaining. With this information, the firing logic can be layered on top of the movement logic because, once again, the state of the firing buttons is included in the user command data structure that is shared between the client and the server. Of course, this can get complicated if the actual weapon logic is different between client and server. In Half-Life, we chose to avoid this complication by moving the implementation of a weapon's firing logic into "shared code" just like the player movement code. All of the variables that contribute to determining weapon state (e.g., ammo, when the next firing of the weapon can occur, what weapon animation is playing, etc.), are then part of the authoritative server state and are replicated to the client so that they can be used on the client for prediction of weapon state there.

Predicting weapon firing on the client will likely lead to the decision also to predict weapon switching, deployment, and holstering. In this fashion, the user feels that the game is 100% responsive to his or her movement and weapon activation activities. This goes a long way toward reducing the feeling of latency that many players have come to endure with today's Internet-enabled action experiences.

Umm, This is a Lot of Work

Replicating the necessary fields to the client and handling all of the intermediate state is a fair amount of work. At this point, you may be asking, why not eliminate all of the server stuff and just have the client report where s/he is after each movement? In other words, why not ditch the server stuff and just run the movement and weapons purely on the client-side? Then, the client would just send results to the server along the lines of, "I'm now at position x and, by the way, I just shot player 2 in the head." This is fine if you can trust the client. This is how a lot of the military simulation systems work (i.e., they are a closed system and they trust all of the clients). This is how peer-to-peer games generally work. For Half-Life, this mechanism is unworkable because of realistic concerns about cheating. If we encapsulated absolute state

[2] ftp://ftp.idsoftware.com/idstuff/source/q1source.zip

data in this fashion, we'd raise the motivation to hack the client even higher than it already is[3]. For our games, this risk is too high and we fall back to requiring an authoritative server.

A system where movements and weapon effects are predicted client-side is a very workable system. For instance, this is the system that the Quake3 engine supports. One of the problems with this system is that you still have to have a feel for your latency to determine how to lead your targets (for instant hit weapons). In other words, although you get to hear the weapons firing immediately, and your position is totally up-to-date, the results of your shots are still subject to latency. For example, if you are aiming at a player running perpendicular to your view and you have 100 milliseconds of latency and the player is running at 500 units per second, then you'll need to aim 50 units in front of the target to hit the target with an instant hit weapon. The greater the latency, the greater the lead targeting needed. Getting a "feel" for your latency is difficult. Quake3 attempted to mitigate this by playing a brief tone whenever you received confirmation of your hits. That way, you could figure out how far to lead by firing your weapons in rapid succession and adjusting your leading amount until you started to hear a steady stream of tones. Obviously, with sufficient latency and an opponent who is actively dodging, it is quite difficult to get enough feedback to focus in on the opponent in a consistent fashion. If your latency is fluctuating, it can be even harder.

Display of Targets

Another important aspect influencing how a user perceives the responsiveness of the world is the mechanism for determining, on the client, where to render the other players. The two most basic mechanisms for determining where to display objects are extrapolation and interpolation[4].

For extrapolation, the other player/object is simulated forward in time from the last known spot, direction, and velocity in more or less a ballistic manner. Thus, if you are 100 milliseconds lagged, and the last update you received was that (as above) the other player was running 500 units per second perpendicular to your view, then the client could assume that in "real time" the player has moved 50 units straight ahead from that last known position. The client could then just draw the player at that extrapolated position and the local player could still more or less aim right at the other player.

The biggest drawback of using extrapolation is that player's movements are not very ballistic, but instead are very non-deterministic and subject to high jerk[5]. Layer on top of this the unrealistic player physics models that most FPS games use, where player's can turn instantaneously and apply unrealistic forces to create huge accelerations at arbitrary angles and you'll see that the extrapolation is quite often incorrect. The developer can mitigate the error by limiting the extrapolation time to a reasonable value (QuakeWorld, for instance, limited extrapolation to 100 milliseconds). This limitation helps because, once the true player position is finally received, there will be a limited amount of corrective warping. In a world where most players still have greater than 150 milliseconds of latency, the player must still lead other

[3] A discussion of cheating and what developers can do to deter it is beyond the scope of this paper.
[4] Though hybrids and corrective methods are also possible.
[5] "Jerk" is a measure of how fast accelerative forces are changing.

players in order to hit them. If those players are "warping" to new spots because of extrapolation errors, then the gameplay suffers nonetheless.

The other method for determining where to display objects and players is interpolation. Interpolation can be viewed as always moving objects somewhat in the past with respect to the last valid position received for the object. For instance, if the server is sending 10 updates per second (exactly) of the world state, then we might impose 100 milliseconds of interpolation delay in our rendering. Then, as we render frames, we interpolate the position of the object between the last updated position and the position one update before that (alternatively, the last render position) over that 100 milliseconds. As the object just gets to the last updated position, we receive a new update from the server (since 10 updates per second means that the updates come in every 100 milliseconds) we can start moving toward this new position over the next 100 milliseconds.

If one of the update packets fails to arrive, then there are two choices: We can start extrapolating the player position as noted above (with the large potential errors noted) or we can simply have the player rest at the position in the last update until a new update arrives (causing the player's movement to stutter).

The general algorithm for this type of interpolation is as follows:

```
Each update contains the server time stamp for when it was generated[6]

From the current client time, the client computes a target time by
subtracting the interpolation time delta (100 ms)

If the target time is in between the timestamp of the last update and the one
before that, then those timestamps determine what fraction of the time gap
has passed.

This fraction is used to interpolate any values (e.g., position and angles).
```

In essence, you can think of interpolation, in the above example, as buffering an additional 100 milliseconds of data on the client. The other players, therefore, are drawn where they were at a point in the past that is equal to your exact latency plus the amount of time over which you are interpolating. To deal with the occasional dropped packet, we could set the interpolation time as 200 milliseconds instead of 100 milliseconds. This would (again assuming 10 updates per second from the server) allow us to entirely miss one update and still have the player interpolating toward a valid position, often moving through this interpolation without a hitch. Of course, interpolating for more time is a tradeoff, because it is trading additional latency (making the interpolated player harder to hit) for visual smoothness.

[6] It is assumed in this paper that the client clock is directly synchronized to the server clock modulo the latency of the connection. In other words, the server sends the client, in each update, the value of the server's clock and the client adopts that value as its clock. Thus, the server and client clocks will always be matched, with the client running the same timing somewhat in the past (the amount in the past is equal to the client's current latency). Smoothing out discrepancies in the client clock can be solved in various ways.

In addition, the above type of interpolation (where the client tracks only the last two updates and is always moving directly toward the most recent update) requires a fixed time interval between server updates. The method also suffers from visual quality issues that are difficult to resolve. The visual quality issue is as follows. Imagine that the object being interpolated is a bouncing ball (which actually accurately describes some of our players). At the extremes, the ball is either high in the air or hitting the pavement. However, on average, the ball is somewhere in between. If we only interpolate to the last position, it is very likely that this position is not on the ground or at the high point. The bounciness of the ball is "flattened" out and it never seems to hit the ground. This is a classical sampling problem and can be alleviated by sampling the world state more frequently. However, we are still quite likely never actually to have an interpolation target state be at the ground or at the high point and this will still flatten out the positions.

In addition, because different users have different connections, forcing updates to occur at a lockstep like 10 updates per second is forcing a lowest common denominator on users unnecessarily. In Half-Life, we allow the user to ask for as many updates per second as he or she wants (within limit). Thus, a user with a fast connection could receive 50 updates per second if the user wanted. By default, Half-Life sends 20 updates per second to each player the Half-Life client interpolates players (and many other objects) over a period of 100 milliseconds.[7]

To avoid the flattening of the bouncing ball problem, we employ a different algorithm for interpolation. In this method, we keep a more complete "position history" for each object that might be interpolated.

The position history is the timestamp and origin and angles (and could include any other data we want to interpolate) for the object. Each update we receive from the server creates a new position history entry, including timestamp and origin/angles for that timestamp. To interpolate, we compute the target time as above, but then we search backward through the history of positions looking for a pair of updates that straddle the target time. We then use these to interpolate and compute the final position for that frame. This allows us to smoothly follow the curve that completely includes all of our sample points. If we are running at a higher framerate than the incoming update rate, we are almost assured of smoothly moving through the sample points, thereby minimizing (but not eliminating, of course, since the pure sampling rate of the world updates is the limiting factor) the flattening problem described above.

[7] The time spacing of these updates is not necessarily fixed. The reason why is that during high activity periods of the game (especially for users with lower bandwidth connections), it's quite possible that the game will want to send you more data than your connection can accommodate. If we were on a fixed update interval, then you might have to wait an entire additional interval before the next packet would be sent to the client. However, this doesn't match available bandwidth effectively. Instead, the server, after sending every packet to a player, determines when the next packet can be sent. This is a function of the user's bandwidth or "rate" setting and the number of updates requested per second. If the user asks for 20 updates per second, then it will be at least 50 milliseconds before the next update packet can be sent. If the bandwidth choke is active (and the server is sufficiently high framerate), it could be 61, etc., milliseconds before the next packet gets sent. Thus, Half-Life packets can be somewhat arbitrarily spaced. The simple move to latest goal interpolation schemes don't behave as well (think of the old anchor point for movement as being variable) under these conditions as the position history interpolation method (described below).

The only consideration we have to layer on top of either interpolation scheme is some way to determine that an object has been forcibly teleported, rather than just moving really quickly. Otherwise we might “smoothly” move the object over great distances, causing the object to look like it’s traveling way too fast. We can either set a flag in the update that says, “don’t interpolate” or “clear out the position history,” or we can determine if the distance between the origin and one update and another is too big, and thereby presumed to be a teleportation/warp. In that case, the solution is probably to just move the object to the latest know position and start interpolating from there.

Lag Compensation

Understanding interpolation is important in designing for lag compensation because interpolation is another type of latency in a user’s experience. To the extent that a player is looking at other objects that have been interpolated, then the amount of interpolation must be taken into consideration in computing, on the server, whether the player’s aim was true.

Lag compensation is a method of normalizing server-side the state of the world for each player as that player’s user commands are executed. You can think of lag compensation as taking a step back in time, on the server, and looking at the state of the world at the exact instant that the user performed some action. The algorithm works as follows:

```
Before executing a player’s current user command, the server:

      Computes a fairly accurate latency for the player

      Searches the server history (for the current player) for the world
      update that was sent to the player and received by the player just
      before the player would have issued the movement command

      From that update (and the one following it based on the exact target
      time being used), for each player in the update, move the other players
      backwards in time to exactly where they were when the current player’s
      user command was created.  This moving backwards must account for both
      connection latency and the interpolation amount[8] the client was using
      that frame.

Allow the user command to execute (including any weapon firing commands,
etc., that will run ray casts against all of the other players in their “old”
positions).

Move all of the moved/time-warped players back to their correct/current
positions
```

Note that in the step where we move the player backwards in time, this might actually require forcing additional state info backwards, too (for instance, whether the player was alive or dead or whether the player was ducking). The end result of lag compensation is that each

[8] Which Half-Life encodes in the lerp_msec field of the usercmd_t structure described previously.

local client is able to directly aim at other players without having to worry about leading his or her target in order to score a hit. Of course, this behavior is a game design tradeoff.

Game Design Implications of Lag Compensation

The introduction of lag compensation allows for each player to run on his or her own clock with no apparent latency. In this respect, it is important to understand that certain paradoxes or inconsistencies can occur. Of course, the old system with the authoritative server and “dumb” or simple clients had it’s own paradoxes. In the end, making this tradeoff is a game design decision. For Half-Life, we believe deciding in favor of lag compensation was a justified game design decision.

The first problem of the old system was that you had to lead your target by some amount that was related to your latency to the server. Aiming directly at another player and pressing the fire button was almost assured to miss that player. The inconsistency here is that aiming is just not realistic and that the player controls have non-predictable responsiveness.

With lag compensation, the inconsistencies are different. For most players, all they have to do is acquire some aiming skill and they can become proficient (you still have to be able to aim). Lag compensation allows the player to aim directly at his or her target and press the fire button (for instant hit weapons[9]). The inconsistencies that sometimes occur, however, are from the points of view of the players being fired upon.

For instance, if a highly lagged player shoots at a less lagged player and scores a hit, it can appear to the less lagged player that the lagged player has somehow “shot around a corner”[10]. In this case, the lower lag player may have darted around a corner. But the lagged player is seeing everything in the past. To the lagged player, s/he has a direct line of sight to the other player. The player lines up the crosshairs and presses the fire button. In the meantime, the low lag player has run around a corner and maybe even crouched behind a crate. If the high lag player is sufficiently lagged, say 500 milliseconds or so, this scenario is quite possible. Then, when the lagged player’s user command arrives at the server, the hiding player is transported backward in time and is hit. This is the extreme case, and in this case, the low ping player says that s/he was shot from around the corner. However, from the lagged player’s point of view, they lined up their crosshairs on the other player and fired a direct hit. From a game design point of view, the decision for us was easy: let each individual player have completely responsive interaction with the world and his or her weapons.

In addition, the inconsistency described above is much less pronounced in normal combat situations. For first-person shooters, there are two more typical cases. First, consider two players running straight at each other pressing the fire button. In this case, it’s quite likely

[9] For weapons that fire projectiles, lag compensation is more problematic. For instance, if the projectile lives autonomously on the server, then what time space should the projectile live in? Does every other player need to be “moved backward” every time the projectile is ready to be simulated and moved by the server? If so, how far backward in time should the other players be moved? These are interesting questions to consider. In Half-Life, we avoided them; we simply don’t lag compensate projectile objects (that’s not to say that we don’t predict the sound of you firing the projectile on the client, just that the actual projectile is not lag compensated in any way).
[10] This is the phrase our user community has adopted to describe this inconsistency.

that lag compensation will just move the other player backwards along the same line as his or her movement. The person being shot will be looking straight at his attacker and no “bullets bending around corners” feeling will be present.

The next example is two players, one aiming at the other while the other dashes in front perpendicular to the first player. In this case, the paradox is minimized for a wholly different reason. The player who is dashing across the line of sight of the shooter probably has (in first-person shooters at least) a field of view of 90 degrees or less. In essence, the runner can’t see where the other player is aiming. Therefore, getting shot isn’t going to be surprising or feel wrong (you get what you deserve for running around in the open like a maniac). Of course, if you have a tank game, or a game where the player can run one direction, and look another, then this scenario is less clear-cut, since you might see the other player aiming in a slightly incorrect direction.

Conclusion:

Lag compensation is a tool to ameliorate the effects of latency on today’s action games. The decision of whether to implement such a system rests with the game designer since the decision directly changes the feel of the game. For Half-Life, Team Fortress and Counter Strike, the benefits of lag compensation easily outweighed the inconsistencies noted above.

The Art and Science of Level Design

Cliff Bleszinski, Epic Games

Originally presented at GDC 2000

It is becoming increasingly difficult to define the role of the team member known as the "Level Designer." Level design is as much an **art** as it is a **science**; it requires artistic skills and know-how as well as an extensive technical knowledge. A designer with tremendous traditional art or architectural experience will not succeed if he cannot grasp issues such as framerate, gameflow, and pacing. A designer who understands these elements yet has no architectural or art experience is doomed to fail as well. Art and Science are the Yin and the Yang of design and it takes the efforts of very talented and dedicated individuals to produce high quality levels.

I. Defining the Role

In the earlier years of the gaming industry, there was no such thing as a Level Designer. Programmers were the "one stop shop" of game creation; they were the ones responsible for designing, producing, and finishing products. With the evolving state of 3d technology, the need for these "digital architects" has appeared, and 3d environments are more gorgeous than ever.

Above and beyond everything that is outlined in this presentation, the role of the level designer on any given project will be defined by two key factors:

What technology will be used for this project?

A project administrator can cut down on training costs and time by hiring talent that is experienced with editing tools that are presently available to the community. For instance, if an Unreal Technology Licensee were to hire talent they'd benefit from acquiring someone who has previously created content with the editor and released it online, or has worked at another technology Licensee. A savvy recruiter will comb map design collection pages as well as closely examining the content produced by peers who are using the same technology for their titles.

What kind of project will we be building with this technology?

Taking a master deathmatch level designer and asking him to create sprawling landscapes for an Everquest style Massively Multiplayer Role-Playing Game would be a big mistake. Even if the designer were able to adapt and create great content then the time and overhead taken to train him in the new design and direction would not be worth the effort. It is possible for a designer who is "trying out" for a job to test his hand at another style in an effort to impress his potential employers, but by the time his content is presentable the job may have passed him by. Although many design elements are universal and will carry over from one style of game to another, it is key to reduce any extra time or risk that is taken in hiring new talent as budgets are constantly rising.

On Ownership

Until recently, at many development studios, there has been a notion of "ownership" in the realm of level design. A level was "owned" by a designer; no one touched his work and he was the one solely responsible for the content. Level Designers would become defensive, even hostile, if another LD suggested modifying his work.

The gaming industry is about evolution. The designers, programmers, and hardware manufacturers who do not evolve quickly fade out and die. The Level Designer is no different from these rules, much like his peers he must evolve. That said, it is no longer possible for one LD to maintain "ownership" of a level as computers and gaming machines are becoming more and more capable of rendering extremely detailed environments. The talent that is hired must be comfortable with the idea of others modifying and improving their work.

There is a direct correlation between the detail that a technology is capable of and the amount of ownership that one designer has over a particular level. With Moore’s law holding true (processor speed doubles every eighteen months) and 3d accelerators constantly raising the bar the detail that game engines are capable of is staggering. It is simply impossible for one driven person to build the necessary amount of detail into level locations in the allocated time, and the more detail technology can push the more people will be required to work on levels.

In addition to having dedicated world texture artists and environment concept designers the need will soon emerge for dedicated "prop" people; artists who create content that will fill up previously static and barren environments. Most architecture is relatively simple, much of the detail in the real world comes from the "clutter," the chairs, tables, and decorations that fill these places up.

Teams may soon see the addition of "scripting" people who are responsible for storyboarding in-game events as well as assisting in the design and direction of these events. A person of these abilities would need cinematic experience as well as excellent knowledge of the tools that are used to create "cinemas," such as a scripting language or editor.

It is very likely that the level designer will be like a chef, taking various "ingredients" from other talented people and mixing them into something special while following the "recipe" of a design document. Right now there are companies that have artists lighting levels, as well as doing custom texture work on a per-surface basis. The level designer will evolve to the role of the glue of a project, the hub at which everything comes together.

b. The Glue

Jay Wilbur once said, "Level Design is where the rubber hits the road."

This quote holds true today, and will continue to hold true in the future. Level designers are quickly becoming some of the most important members of a development team.

Nine times out of ten one finds that programmers are the bottlenecks on a project. A game is not supposed to ship until it is clear of all "A" class bugs, and this requires much programming gusto to clean up and ship a game. On many projects this bottleneck will eventually slide into the realm of the level designer as they're where the "rubber hits the road." The LD is the one who is taking everyone else's hard work and tying it together into a cohesive package. The designer takes the textures created by the artists and places them on his level geometry, or asks an artist to create custom work for his level. He'll figure out where and when to place hostile AI that was created by programmers and 3d artists while all of it is being rendered by the work of the engine programmer.

A level designer is not just an architecture monkey or a guy who throws "cool stuff" into the pot of development. Above and beyond everything else they need the ability to judge what is *fun*, what gameplay elements work and what do not. He needs to judge what content works in any context while making sure his work is cohesive with the rest of the game.

II. Design Commandments

Now that the role of the level designer is defined the following are some tips for him to live by.

Designer, Evaluate Thyself

The best level designers are never afraid to step back and re-evaluate their content. Often this requires a period of respite from the work in question; distance can clear up a clouded mind. A great designer isn't afraid to throw content out or re-work a concept that needs attention.

It is also extremely important for a level designer to recognize when he is becoming tired of his own work and when his work is not coming together. There is a huge difference between the two; in one instance a designer becomes weary of playing his own content over and over and is just sick of it. A great level might get scrapped or reworked because a development cycle is dragging on and a designer feels the work is not as fresh as it used to be. The designer must recognize that his view is tainted; he has been playing this content for months on end and by nature the work becomes stale to him. This does not mean that the work will have any less impact on the user, however! At this point, a designer should have his map tested repeatedly by new and experienced players and simply polish the work instead of reworking it.

Thou Shalt Seek Peer Criticism

Assembling and maintaining a great team of designers is a challenging task. It is important to hire easy-going talent that gets along well together. A great designer is never afraid to take criticism from his peers; in fact, a great designer is the sum of himself plus his peers. Many artists feel that they’re more talented than the next, this cockiness can be the weak link in a design team. The ideal designer seeks criticism even from those he may consider "less talented" than he, because even if he believes that the critic in question has no skills the commentary will be fresh and from a new perspective. The best way to go about doing this is to have periodic "peer evaluations" where a lead designer or lead level designer picks two designers and has them evaluate each others work while acting as a mediator.

Thou Shalt Value Rivalries

In addition to taking suggestions from one another it is key for level designers to feel a desire to "one up" each other. Healthy competition in any area of a development team means improved results. However, a positive, healthy competition can quickly turn ugly as one designer may accuse another of stealing his style or designs.

If a designer is emulating the style of another this benefits the project, as the environments will become more consistent. The Design Lead should encourage overlapping designs and work towards smoothing ruffled feathers and the Art Director should make sure the environments are consistent by leading the team’s aesthetic designs.

Do Thy Homework

As much of this work is moving into the realm of the Art Director and art team, the designers remain the Digital Architects and they will still be responsible for much of the look and feel of the levels. Therefore, if a project calls for an accurate Roman Empire then everyone had better be doing his or her homework. Having a shared directory of R+D images on a server as well as an art bible that is referred to all designers and artists will contribute to a more consistent look and feel.

At any given point in development a designer needs to be able to step back, look at his work and think, "Does this make *sense?*" More often than not he’ll discover little details that make no sense, such as structurally impossible architecture that seems out of place or ice beasts near lava pits. The users may not notice these details on the conscious level but will sure feel it on a subconscious level which will affect his overall game experience negatively.

Good research lends itself to good planning. Some designers simply sit down and build while others carefully plan every nook and cranny of the game. The best designs are the ones that are a combination of careful on-paper planning and improvisation.

Thy Framerate Shall Not Suck

If the designers are working with a technology that can push 100 million polys then they’re going to try to make the tech look like it can push 3 times that. Although much of the framerate issue falls upon the programmers, with optimizations and level of detail technology, it is extremely important that designers have hardcoded guidelines for framerates, detail levels, and RAM usage.

The Lead Level Designer should be the one responsible for enforcing hardcoded design limitations. Unreal Tournament had extremely strict limitations on how detailed a level’s geometry could be, as well as overall framerate time.

Framerate can be sacrificed somewhat if a title is slower-paced and does not require action-oriented reflexes. However, if the team is building an action game and levels are bloated and framerates are dying then the hardcore action users will reject the title and every review will read "looks nice, runs terribly."

Thou Shalt Deceive

Pay no attention to the man behind the curtain. If a designer can simulate a newer technology with some trickery then by all means allow and encourage this. If the programmers are exclaiming things such as "I don’t remember programming that!" or "How did you do that?" then something special is going on. If a scene can look more detailed with creative texturing then go for it. If bump mapping or specular highlighting can be faked even though the engine does not "truly" support it then why not? Only the hardest of the hardcore gamer will know the difference.

This mode of thinking can be carried over to nearly every aspect of development, not just level design. Programmers can find creative ways to "fake" new effects; a sneaky artist can make a character look more detailed than he actually is with good texture mapping or smart use of polygons.

A very basic example of this would be a designer who uses "masked" textures to create the illusion of much more detailed geometry. For example, making a grate on a wall can be done by only using one polygon that’s masked instead of constructing the actual holes of the grate out of individual level brushes. Masked and translucent surfaces were used in many areas of Unreal Tournament to simulate weather effects such as snow and rain.

III. On Design Techniques

Many of these techniques can be applied to either a single player oriented title or a multiplayer oriented title. Every technique is governed by an overall concept of "Gameflow." It is the mystical "life force" that makes a good game fun and it is very much a reward-response system that challenges the gamer and then provides a "treat" for completing tasks. Time and Time again this document will refer to the "Carrot on the End of the Stick." This is the incentive for the gamer to keep going; many of these "carrots" are built and planted by the level designer as these drive gameflow.

The Gameflow of a Single Player title is driven entirely by the level designer. He's the one who is creating a task and then placing the carrot in front of the gamer, encouraging him to complete the task. It is very much a cause and effect design, create a problem and then encourage the player to solve it. This is why many titles use violent elements as their focus as it is the easiest and most basic form of conflict.

Multiplayer Gameflow varies quite a bit from Single Player Gameflow; it is more about rationing risk and reward in a social environment. A Level Designer who is building for a Multiplayer-oriented title is much like a playground architect. He's building the space where real people will be driving the game and experiencing the action firsthand; the gamers themselves largely dictate the gameflow. Designer-placed elements such as AI or story can often prod this along but more often than not it is the gamers who are the catalysts that keep that carrot on the end of the stick for the gamer. A title like Ultima Online creates a world where designers carefully place resources around the player and the users who harvest these resources are proud to wear their spoils of war. This creates a desire for the "have-nots" to become rich and prosperous and drives the game.

Controlled Freedom

Let the player think he has a choice in where to go and what to do but gently guide him to his destination.

This is an avidly debated topic; if a player has the freedom to go anywhere and do anything (as many gamers *claim* they want) then he will quickly get lost and frustrated. By keeping level design somewhat linear and giving the illusion that there are multiple paths one has the freedom to choose then the player will have a more enjoyable play experience. This way, the player experiences the best of both worlds; the player gets to the carrot on the end of the stick, and feels like he made the right decisions on where to go.

This can take more time to design but ultimately adds up for a more enjoyable single player experience for the user. It is completely possible to build a title that revolves around the notion of "go anywhere, do anything" but a developer who does this must allocate plenty of time and funding to make this a reality. Often previous titles that have had very much "open-ended" designs have had users that have found themselves lost and asking "what do I do next?" Only the most hardcore of the hardcore gamer will stick with a title that is too open-ended. It *can* be done; it simply requires longer design times and a more focused and dedicated user.

Pacing

Constant scares dull the senses.

The scariest horror movies are the ones that lull the viewers into a false sense of security and then spring something scary upon them, and a great level is no different. An excellent recent example of this is System Shock 2. One minute the player is being chased down by pipe wielding maniac hybrids, the next he's tucked away in a quiet bedroom aboard the Von Braun, reading log files from dead crewmembers while wondering what will be around the next corner. If the monsters were constantly in the player's face the game would cease to be scary. However, the down time lets the player forget, for a moment, the peril that he is in… just long enough so that his guard drops and he's scared (and killed) by the next baddie.

The best multiplayer titles are driven by a system of good pacing through intelligent resource distribution. It requires a bit of effort in one's skills or "character" to improve at the game and move up in the online world. For instance, in a Deathmatch or Teamplay game a player acquires his skills by learning battle arenas and how to aim. A designer's pacing in a level will determine if the game works or not, if every character starts with a crazy arsenal or if there are areas that are impossible to breach or defend then the game suddenly fails to be entertaining. In titles that are more role-playing driven or strategy oriented a designer must be cautious with wealth and resources. If, for example, there isn't enough ore to mine in an adventure game then players cannot build weapons and the game system crumbles. The designer is the key to making the entire game system work and often has to work and rework his ideas to make sure they balance the world well.

Risk Incentive

In single player design, there are oodles of ways a designer can utilize this time tested technique to let the gamer make his own decisions about how much trouble he's going to get himself into for treasure.

That's the beauty of risk incentive. The player weighs the risk; he assesses the challenge, and gets to make a decision. He feels like he's in control, and the designer provides him with a choice.

For example, in a traditional shooter the designer might place ammunition or health below a pair of sentry turrets. The turrets can easily be avoided by crawling behind a pair of desks, however if the player wants to make a dash for the goodies it is *his choice*. Therefore, if the guns rip him to shreds and he screws up he blames *himself*, not the designers.

In a Deathmatch style game a player will have the choice of going for an ass-kicking weapon, only if he risks his neck by going into an extremely open and well guarded spot.

Never underestimate the usefulness of this technique.

On Revisiting

The concept of "Revisiting" or "Doubling Back" refers to the gamer seeing an inaccessible area of a level and wondering "How do I get there?" The gamer then proceeds to complete a series of tasks which move the game/story along (as well as his virtual self) and he then suddenly looks around and realizes "Oh! I'm up there now!"

Revisiting areas from a different angle is a good thing for designers to practice. It keeps the gamer motivated as he tears through your designs, as well as saving time and money. The same rooms are viewed from multiple angles as well as revisited, and this saves the designer from building more areas. This will be more and more of a blessing as levels become more detailed and expensive to produce in the near future.

Many Multiplayer titles are dependent upon revisiting areas of levels as Multiplayer design often focuses around character interaction instead of just exploring. A recursive design is extremely important in any kind of social title.

Supply And Demand

Leave the gamer always concerned about running out of ammunition and/or health, but not to the point where he's running around bullet-less, dying constantly, while cursing the designers and their product. This is yet another carrot on the end of the stick trick that makes for a satisfying gaming run. It teaches resource management, and makes it a better experience when the gamer finds health and ammo. Good supply and demand makes these goodies more valuable.

In a Multiplayer Title the designer has to account for players trying every available option to exploit the game and level design. It is key for the designer to balance the amount of resources that are available in *any* multiplayer game to prevent a tiny percent of the playing population from enjoying all of the virtual "wealth."

Scene Composition and Contrast

Relatively simple objects arranged in an interesting method can result in a far more eye-pleasing image. This is true with art, architecture and, of course, level design. It becomes especially relevant when working with low-polygon geometry and strict detail budgets.

Many art classes will spend time focusing on the idea of scene composition. This is another example where an art background will come in handy for a designer.

Work With The AI Guy

AI is tied directly into the structure and composition of a level. It is where the AI does its thing, it is the place where all that hard work on the part of the AI guy is supposed to be shown.

It is crucial for a level designer to construct areas that take advantage of the AI while working with the AI guy and figuring out what the AI is going to do. For instance, if there is an AI that is really good in firefights, ducking behind boxes and taking pot shots at the player, a designer should plan to build an environment with waist high crates all over the place. If the AI guru programs a great pack AI, make space that accommodates it. Often AI does not work perfectly, it is important to maintain patience and have faith in the AI talent as the designers manage to iron out kinks in the system.

Smart designers and programmers will work together to create memorable scenes where puzzles and areas are built around crafty artificial intelligence.

On Sound

Steven King, in *Danse Macabre*, said something along the lines of:

"When the lightning crashes and the door opens and you see a ten foot bug standing there, a part of you sighs and thinks "Whew, I thought it was going to be a **TWENTY** foot bug."

Designers must work closely with sound technicians to assure a compelling and exciting audio experience. A great designer never underestimates how much mileage he can get out of a good bump in the night. No matter how good the talent is, the monster that is in the gamer’s head is always scarier than what is seen onscreen. If the title calls for chills and thrills, let the sound do much of the work!

Intelligent Backtracking

If a designer is forcing a gamer to backtrack he must make sure that it is done in a logical and non-frustrating manner. This is a dangerous time in design, as the "carrot on the stick" of seeing a new area is gone. A designer is re-using a previously seen area and it is important to make the area seem fresh or interesting as the player navigates it. This often requires subtle scene changes, or the addition of new hostiles to prevent the area from seeming "dead" and "used." Much like a used-car dealer will polish up an older model, a designer who is re-using an area must put more effort into it to make sure that it seems new and fresh.

It is also key to make sure that the gamer does not get lost as he is backtracking. If, for example, a gamer must activate a pump so he may drain an area with waste and cross then the route back to the previously hazardous area had better be pretty easy to navigate in reverse. Using **controlled freedom** here will ensure that the gamer knows where he’s going; perhaps by blocking off a redundant area or placing highly visible signs that direct him on where to go he’ll have more fun.

IV. The future

Gaming continues to evolve and is heading in a variety of directions, and level designers will be at the forefront of this revolution. Programmers will be responsible for entire tool sets that the designers use and it will be important to have a good synergy between designer and coder.

On Editors

Level Editors will become closer to high end modeling packages such as Lightwave or 3dstudio max, as real time scenes are approaching pre-rendered ones. Many developers have forgone traditional in-house level editors for packages such as these, so it can’t hurt for a designer to learn Max, Maya, or any of these programs. Chances are, in-game editing tools will be at a similar level of complexity in the future.

There are basic 3d editing concepts that these programs are built upon that everyone should know, a designer should understand how to manipulate low polygon geometry as well as high polygon geometry.

On Texturing

In the past, geometry was extremely simple and nearly all of the world detail was done in the textures. Many current titles feature approximately a 50/50 ratio of texture detail to world geometry detail, levels feature many custom textures, a simple polygonal arch will be framed by a custom texture that makes it look that much more detailed and planned. The real Next-Generation titles will feature a more detailed "material" system where simple maps are mixed to create realistic surfaces. For instance, a designer will be able to specify the shininess, depth, and color of any material that will then be placed on world geometry.

On AI

In the future Designers will have to work even more closely with the in house AI programmers. On one hand, design will become easier as AI will become better at tasks such as navigation and conflict, while on the other hand the job will become trickier as users demand more and more cinematic experiences. Smart designers will build many custom AI "scenes," such as exciting stand-offs between hostiles and teammates while building in a failsafe "backup" AI that keeps the scene convincing if the user "breaks" the action. By "breaking" the action the user may, say, blow up a character that is supposed to jump through a window or trigger some sort of action.

On Skills

Every level designer at Epic Games has primary and secondary duties. Some designers, besides working on levels, are competent texture artists. Others are good at modeling characters or decorations. As the level designer evolves it will become more and more important for him to be familiar with many of the tools that artists and 3d modelers use.

This reduces any potential "middle man" time risk. For example, if a texture artist has created a great brick pattern for a designer before leaving for the day and the texture does not tile horizontally on a surface correctly the designer can open the art in PhotoShop and make it tile himself.

A designer who can "do it all" is both a blessing and a curse. If he can create his own textures, architecture, lighting, and decorations then he’s an all-in-one package, a one-man design machine. However, designers who are this talented often have their own notions about how they want "their" work to look and can be very difficult to work with when the time arrives for "shared" design. Another problem with a "do it all" designer is that his time is divided between texture creation, world creation and decoration creation and often finds that he’s bitten off more than he can chew! These designers often require the least amount of management at the start of a project and the most at the end when they’re struggling to finish all they’ve started.

Conclusion

The last quarter of a game’s development cycle is the most crucial for the entire team, especially the level designers. Features that are often broken will finally be working and the game can be solidified and polished. Truly talented designers will shine in these moments.

It is important to remember that, much like many sports, creating a game is not a one-man show. As important as level designers are for the team, they are nothing without quality programmers and talented artists to back them up, and vice-versa.

The gaming industry is constantly evolving; a short while ago there was no such thing as a "level designer" and now they’re key team members on a project. The level designer needs to understand where he fits in amongst the other talented people he works with, and needs to have an open mind and a good artistic sense if he’s going to help put everyone’s hard work together into a fantastic product.

by
Cary Sandvig (Cary.Sandvig@disney.com)
and
Jesse Schell (Jesse.Schell@disney.com)

Introduction

Panda3D (Platform Agnostic Networked Display Architecture) is an open source software architecture for rapid development of networked real-time 3D applications. Its key features are:

- Powerful scenegraph semantics
- Abstraction layers for portability
- Rapid prototyping enabled through late-binding scripting languages
- Easy-to-use network architecture for rapid creation of shared worlds

Panda3D is available for download from www.panda3D.org.

Scenegraph Design

The design of Panda3D was driven by desires for both the ability to capture the semantics of a model cleanly, and to have a thin abstraction for the actual hardware. Many of these decisions were realized in what we call “noun-verb separations,” an attempt to avoid mixing actions or behaviors with static data objects.

This led to a departure from standard practice in the scene-graph design. Commonly, information such as local transformation, color, and texture is stored directly on the nodes of the graph. This typically leads to either very large nodes (to accommodate all possible state changes that may be desired), or a very large number of node types (to provide all the combinations of state change). Instead, we view the arcs of the scene graph as first-class objects which represent some change in state. The nodes, then, represent being in a particular state. Our nodes become very light-weight, holding only actual geometry, or acting as a handle to deal with higher-level structure.

This approach not only simplifies our representation, but also facilitates scene graph inquiries that were difficult before. For instance, a standard request asks how two nodes in the scene graph are related to one another spatially. Now we can ask about such relationships over any state change, such as color, texture, or fogging. This structure has also given us, for example, an extremely flexible instancing mechanism that allows us to alter much more than just the transformation leading to each instance.

The Panda3D scene graph is a specialized instance of a general-purpose graph representation and handling mechanism. Often, many special purpose graphs will exist within a single application, and some nodes may participate in multiple graphs. For example, a sample application in the Panda3D distribution constructs a “data graph” which describes and controls how input from the user is transformed into camera movements and actions on the scene graph. Alternatively, a culling pass can be made on the scene graph, building a new graph on those nodes that represents only what is visible, thus reducing what the drawing pass must traverse.

Other Abstraction Layers

Another part of the design that was guided by the “noun-verb separation” principle is in the Graphics State Guardian (GSG). The GSG plays multiple roles in the Panda3D system: abstraction of the rendering hardware, filter for unnecessary or wasteful state change, and “verb” for drawing the “noun” of the scene graph. This factoring enables the use of many different back-end rendering libraries, even in the same application. New implementations of the GSG can be added without ever having to touch scene graph code. Furthermore, the GSG specifies exactly what functionality needs to be present (or emulated) on a given platform.

Similar thin abstraction layers exist or are in development for other functions, such as: sound, networking, process/threading, windowing, device I/O, etc. Since Panda3D only calls these functions via abstraction layers, third-party libraries (VRPN or NSPR for example) can be used by Panda3D. These libraries are wrapped with an adapting interface. In this way we can leverage the work done in these areas without having changes in these packages affect the Panda3D code that uses it.

Interactive Scripting

Efficiency and rapid prototyping are two keys to creating high quality virtual worlds. Unfortunately, no single computer language adequately provides both. C++ is the coin of the realm for efficiency, and for access to third-party API’s. However, it is an early-binding compiled language, which prevents programmers from making changes to their software while a simulation is running.

Late binding "scripting" languages, such as Scheme, Python and Smalltalk are excellent for rapid prototyping, but lack the necessary efficiency and API access necessary to create detailed virtual worlds.

Interrogate

Panda3D gives the programmer the best of both worlds by providing a way to use both C++ and an interactive scripting language simultaneously. Any language that has a foreign function interface allowing calls to C libraries can make use of Panda3D's Interrogate system.

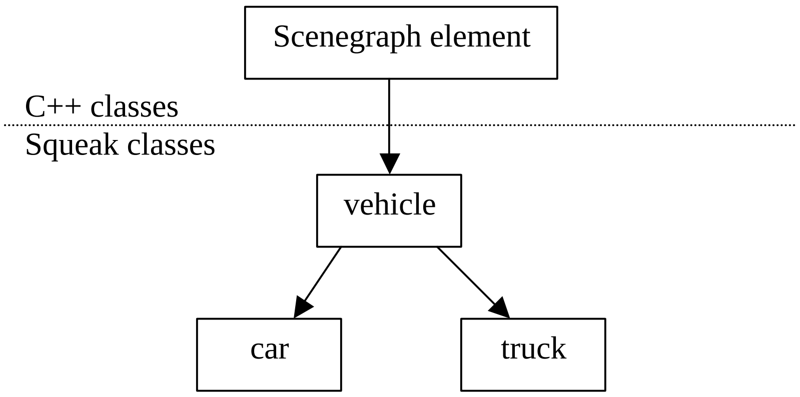
Interrogate works like a compiler, scanning and parsing C++ code. Instead of creating object libraries, it creates an "interrogate database" of objects, methods, global variables, etc. that are contained in the corresponding C++ library. This database may be queried to discover these functional elements and their interfaces.

To make use of this database, one creates an automatic code generator in the scripting language that scans the interrogate database and generates scripting language wrapper calls that execute the C library calls, via the scripting language's foreign function interface. A benefit of this system is that interrogate imposes no restrictions on exactly how a scripting language interfaces with the C libraries, allowing them to interface in whatever way best "fits in" with the natural environment of a particular language.

Squeak

Panda3D includes an automatic code generator for Squeak (an open source Smalltalk implementation: www.squeak.org) because it is our favored scripting language. Squeak is an entirely object-oriented language, although its syntax, rules, and implementation are vastly different from C++. Nonetheless, we have been able to create a Squeak interface to C++ that allows object-orientation to cross the boundary between the languages. For each C++ class, a "proxy class" is automatically generated in the Squeak code, which exposes all of the public functions of the C++ class.

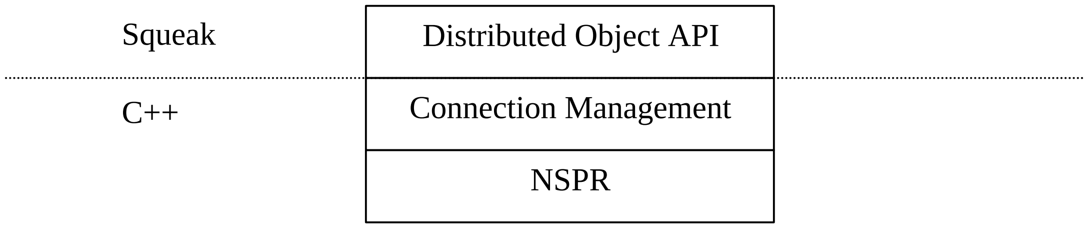
This not only allows for easy access to (and debugging of) C++ classes, but also allows for Squeak methods to be added to C++ classes, and for Squeak classes to appear to inherit from C++ classes. For example, a 3D driving simulation might include vehicle classes which inherited from the "scenegraph element" class. The cross-language inheritance hierarchy might look like the following:



This is particularly useful, since each car or truck has all the interfaces that the efficient C++ scene graph provides, seamlessly integrated with the flexible, reprogrammable-on-the fly interfaces that Squeak classes provide.

Rapid Prototyping of Shared Worlds

The Panda3D support for rapid prototyping of shared worlds is implemented through three layers:



From bottom to top, these are:

NSPR: The Netscape Portable Runtime layer gives portable, robust, open source interface to Posix threads and TCP/UDP sockets. See http://www.mozilla.org.

Connection Management layer: A robust set of C++ classes forming an API that hides most of the details of sockets, byte streams, and threading, while giving the programmer the flexibility to manage arbitrary sets of TCP/UDP connections, and handle them in either single or multi-threaded fashion. Efficient clients, servers, and peer-to-peer solutions can easily be crafted on top of the Connection Management layer, either in C++, or in a scripting language via interrogate.

Distributed Object Layer: A simple system implemented in Squeak for rapid prototyping of shared virtual worlds. At the time of this writing (March 2000), the Distributed Object system is client/server based and is used as follows:

1. **Setup**
    a. *Start a server*
    The server can run on any machine (including a client machine). Panda3D includes a server implemented in Squeak and C++ that is useful for rapid prototyping because it makes it easy to debug problems between clients.
    b. *Create Distributed Object descendant classes*
    Any Squeak class which you would like to exhibit shared behavior needs to inherit from the DistributedObject class. It also needs to specify exactly which methods on it need to be distributed, what types they take, and in what manner they are to be distributed. This is a simple matter of configuring a table of data in one of the methods inherited from DistributedObject.

2. **Startup**
    a. *Create a ClientRepository*
    The ClientRepository is the class that interfaces to the Connection Management layer, and keeps track of all the Distributed Object instances currently in the system. In creating one, you specify all the DistributedObject descendant classes that you plan to use in the simulation, so that the ClientRepository can register them with the server.
    b. *Wait for Server Registration to complete*
    After the ClientRepository registers all the DistributedObject descendant classes with the server, the server then sends instructions to the ClientRepository about what instances of the DistributedObjects already exist in the world, and what their current state is. When this is complete, an event is thrown in the client that says that generation of new DistributedObject instances can now begin.

3. **Generate**
When you generate an instance of a DistributedObject, the ClientRepositories on other clients are automatically notified, and automatically create duplicate instances of the DistributedObject.

4. **Update**
When you update an instance of a DistributedObject, the ClientRepositories on other clients are also automatically notified, and update their local instance to reflect your changes. Updates can happen via TCP (guaranteed to arrive), or UDP (not guaranteed to arrive) as the programmer sees fit. The programmer may also separate the acts of the local update and distributed update to handle cases where, for example, it is desired that local updates happen very frequently, while the sending of remote updates happens only occasionally. Finally, the programmer may also specify whether a particular update is state-based (like a positional coordinate), and should be retained by the server, or event-based (like a chat message) and should not be retained by the server.

5. **Destroy**
Destroying an instance of a DistributedObject causes the remote instances to be destroyed as well, and for the instance to be removed from the server. In addition,

if the server loses its connection with a particular client, any DistributedObjects that it has registered ownership for are automatically destroyed and removed from the database. To create an object that will not be destroyed in this way, a client must transfer ownership of it to the server.

6. **Locking**
   Every distributed object has a semaphore lock on it that controls who currently holds it. Locking is managed by the server, but not enforced. Enforcement is up to the clients. Any client may update any distributed object at any time, but to avoid conflicts in updating, a programmer may choose to use the locking mechanism so that clients can check whether it is acceptable for them to update a particular DistributedObject before doing so.

7. **Zones**
   A simple form of interest management works through numbered zones. When connecting, a client specifies what zone it is in, and the server only sends relevant generate and update commands. When a client changes zones, the server sends "disable" commands to that client for all DistributedObjects in the former zone, and "enable" commands (followed by necessary state updates) to that that client for all DistributedObjects in the new zone. In the current (March 2000) implementation, "disable" maps directly to "destroy", and "enable" maps directly to "generate". Future implementations are expected to include more complex caching schemes.

Conclusion and Future Work

This simple implementation of networking features has been very useful to us for rapidly prototyping shared virtual worlds. Other features currently in development include:

- A scalable server implementation with more complex interest management
- Peer-to-peer DistributedObject support
- Compressed and encrypted data streams
- Advanced streaming features to reduce bandwidth overhead

The features described here, as well as other features (such as soft- and hard-skinned character support, multi-pass rendering support, a real-time shader framework, etc), are the result of our work on previous systems used in the production of theme park and Location Based Entertainment (LBE) VR attractions. We have kept those ideas and tools that worked well, iterating on them to produce the current design. While there is certainly a cost associated with parts of the system, performance is very good and the increased flexibility afforded by the design offers an excellent trade-off.

For more (and more current) information about Panda3D, visit www.panda3d.org.

Excerpts from

3D Game Art f/x and Design

By Luke Ahearn
Coriolis Press, 2001

The following three chapters are excerpted from the soon-to-be-published book, *3D Game Art f/x and Design* by Luke Ahearn. These chapters are still in the pre-production stages and are therefore still in a draft form, but are essentially what will be contained in the final published text. Note, image quality is necessarily degraded for the printed version; you can see color images on the SIGGRAPH course CD or in the published book.

Book Overview:

3D Game Art f/x and Design covers the technology of game graphics from the artists' point of view. The organization and creation of such graphical elements as the interfaces, menus, textures, and games levels or spaces are all discussed. This book is not geared towards helping the artist understand graphic technology, as much as it is focused on helping them understand how to use it. Upon completing this book, the reader will have created all of the 2D assets needed for a 3D game, and will also have assembled them into a running game demonstration.

Author Biography:

Luke Ahearn has authored several books and articles for the computer game industry including *Awesome Game Creation: No Programming Required*! and *Designing 3D Games That Sell!* (both published by Charles River Media). Luke founded Goldtree, a computer game development company, where he designed and developed several game titles including Dead Reckoning and Sorcerer. Currently, Luke is serving as Art Director on the Army Game Project at the Naval Postgraduate School in Monterey, California. You can learn more about Luke or his publications at www.goldtree.com.

Chapter 2

Game Textures: The Basics

No matter how detailed or complex your texture set and your world may get, you will always go back to basics, so you will need to have a good grasp of quickly creating the basic surfaces of most worlds: wood, stone, metal, and more.

Why Do You Need Base Textures?

Although your typical commercially released computer game can contain several hundred textures in each level, you will see that only a few base textures cover the common surfaces of any given game world: floors, walls, ceilings, and large redundant surfaces. This is done for various reasons, and the two biggest reasons are as follows:

* *World consistency* -- Most worlds have a consistent wall, floor, and ceiling texture. A castle, for instance, may have walls of rough-cut stone, floors of polished stone, and ceilings of wooden planks.
* *System requirements* -- Most computer game developers work under an ever-present set of limitations. Using a base texture reduces the number of textures loaded for the game.

  Note: Please note that the final game-ready file size will be relatively small, but the working versions of the textures as unflattened, high resolution Photoshop files can get quite large.

So, although a game world may contain hundreds, or even thousands, of textures, they are all composed of base elements and materials such as wood, stone, or metal. That is what we will be looking at in this chapter.

> **Note:** The term level is typically used in a computer game to denote a division in the world. In a racing game, the level may be called a track, in a sports game an arena, and so on.

What Are Base Textures?

Base textures are the set of textures that will cover most of your game world. In an exterior scene, this may be the ground and sky, even a base tileable tree bark for the trees. In an interior scene this is usually the floor, walls, and ceiling. Base textures generally refer to common surfaces such as wood, ground, stone, and so on. Base textures can also be surfaces that are common in your game world, but not in the real world, such as lava, poisonous slime, solid gold bricks, and more. A base texture is the common denominator of surfaces in your world.

In general, base textures are plain, tileable, and representative of the world, story, and other game elements.

Determining the Base Textures of a World

As a game artist, you need to be able to look at a scene, whether real world or fantasy, and extract the base texture set needed to cover that world. You must develop the ability to look past the lighting, ornamentation, special effects, and even bullet marks, bloodstains, and dirt to see what base elements the world is created from. In real life, you can think of it this way: If you were to empty your room, you would more easily see the simplicity of the surfaces of the floor, walls, and ceiling. Look at the wall in even lighting after a fresh coat of paint, and that is the base texture for that wall, probably a subtle texture at best. Maybe the floor is composed of wooden boards, tiles, or a carpet. Remove the stains, dents and other wear and tear from the furniture, and evenly light the room, and that is the base texture for the floor. Figure 2.1 shows a typical realistic scene from a real castle. Notice the walls of stone, the roof, the water, and the sky.

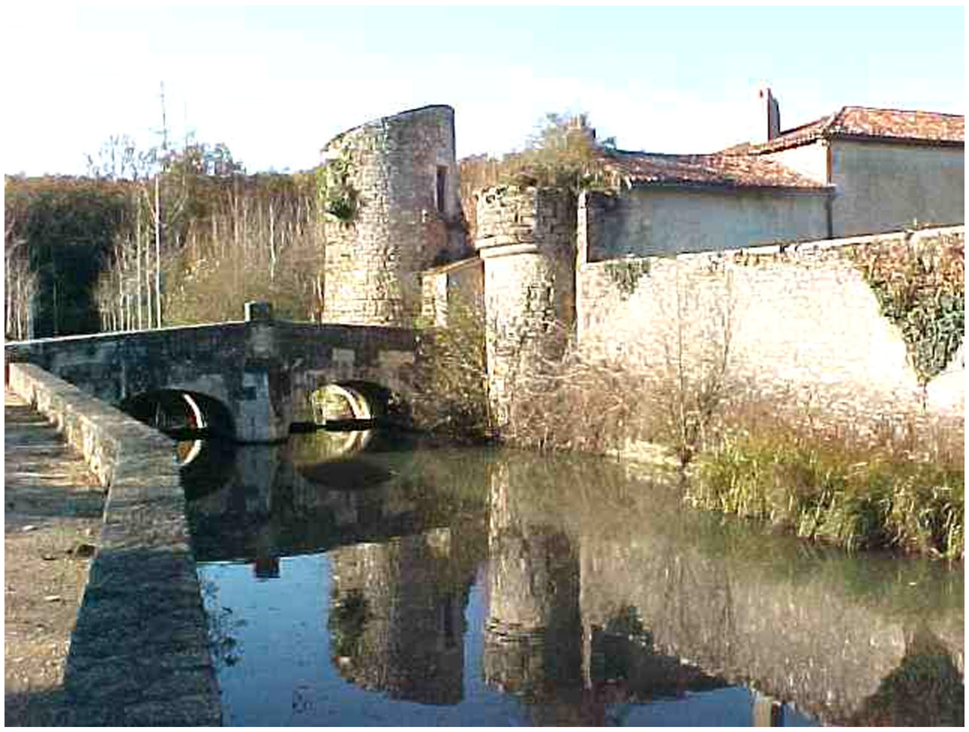


Figure 2.1 This is a real-world scene of a castle. Notice the base textures, the large areas that make up the world.

In Figure 2.2 you see the raw images used to create the stone for the walls and other base textures. Notice that the stains, seams, foliage, and other distinguishing features of the surfaces have been removed.

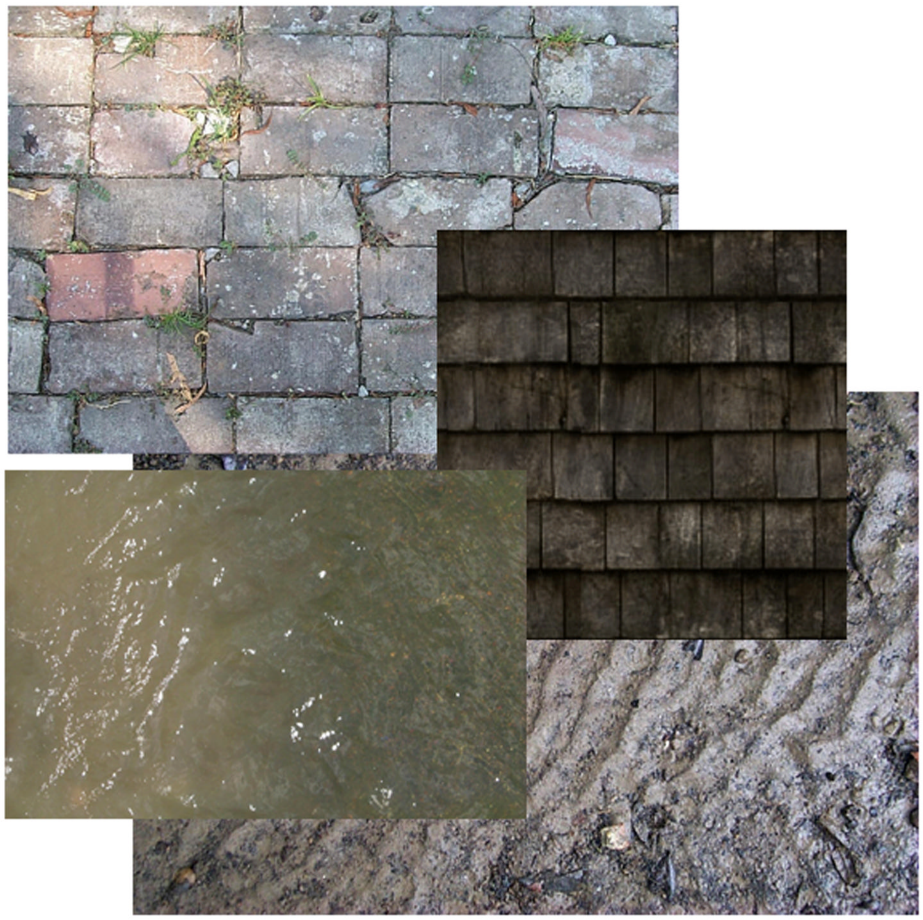


Figure 2.2 The set of base textures and images we will use to re-create the real-world castle scene.

If you look closely at Figure 2.3 and compare it to figure 2.2 you may be able to see the differences between the images. You may first notice that all the images in 2.3 are now square and that they have been cleaned up. Look at the dirt for example, the same dirt base and final image was used in figure 1.1 in the previous chapter to illustrate good and bad tiling of base images. You can see that this texture set is a completed game-ready set of textures that will be used in the level editor to create this real-world scene in a game engine. Notice that the textures fall into categories; there are base textures for the walls, the ground, the roof, and even the water in the moat.

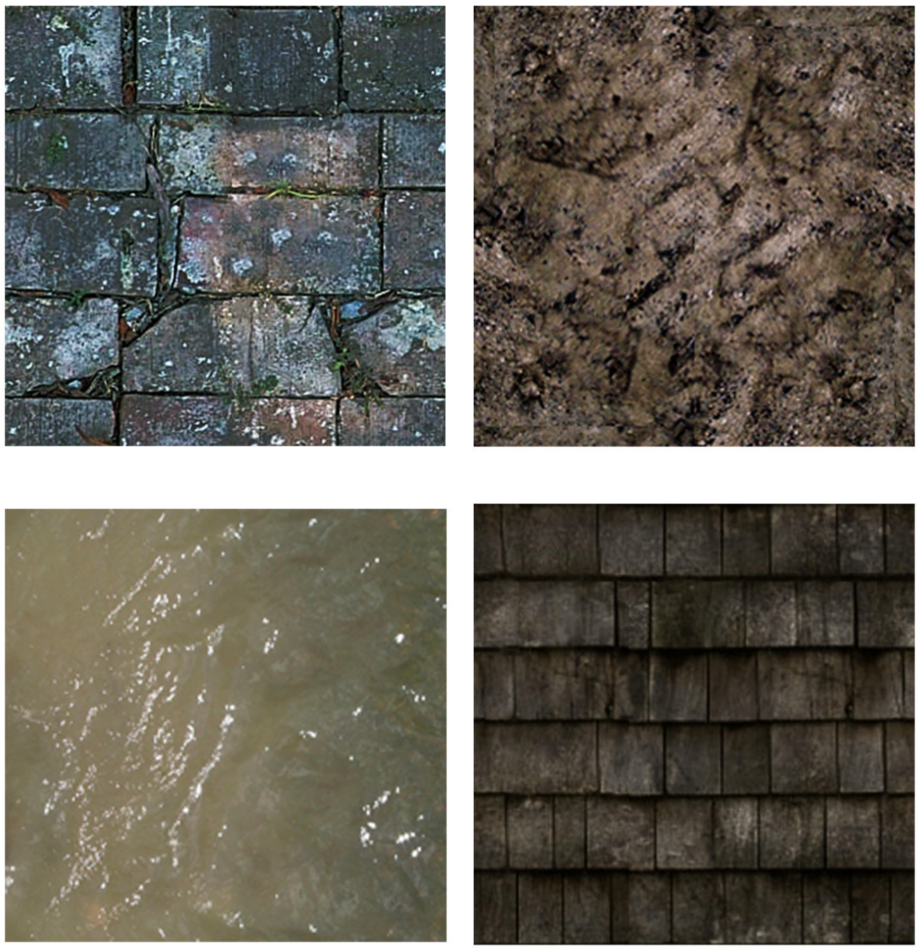


Figure 2.3 The texture set of game-ready images we will use to create this scene in a game.

Finally we are ready to build a scene using basic shapes and the base textures. In Figure 2.4 you can see that the scene in the game world looks like a Hollywood set of the real world scene. We haven't added any details yet such as lighting or ornamentation. Notice the difference in Figure 2.5 when we do add lighting and some ornamentation to the scene.

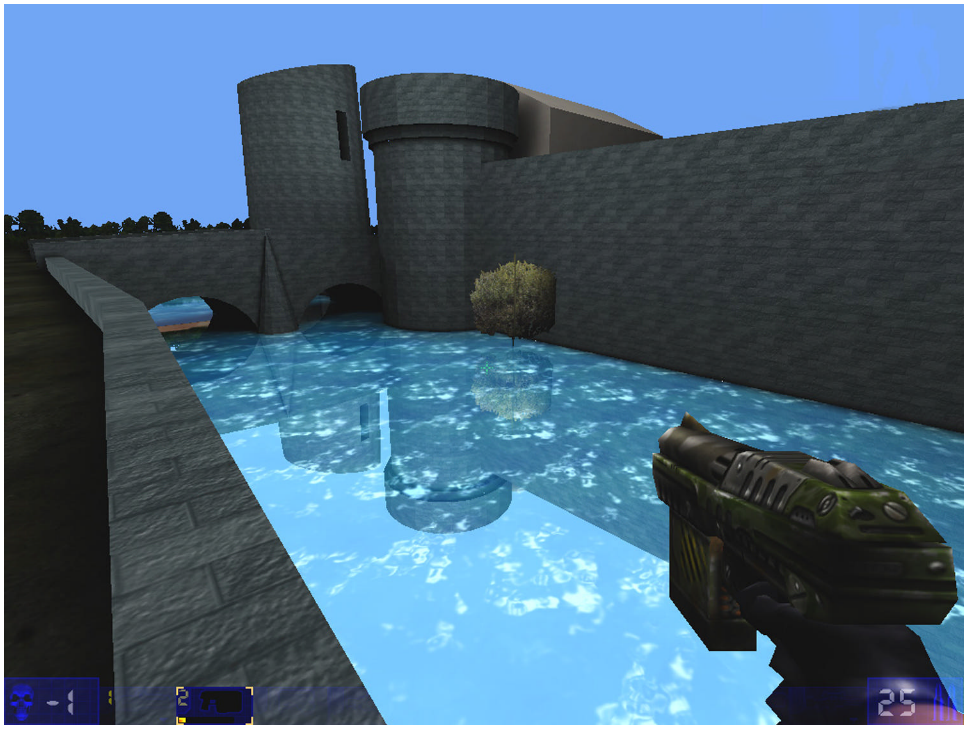


Figure 2.4 A screenshot of the real-world scene built only using basic shapes and base textures.

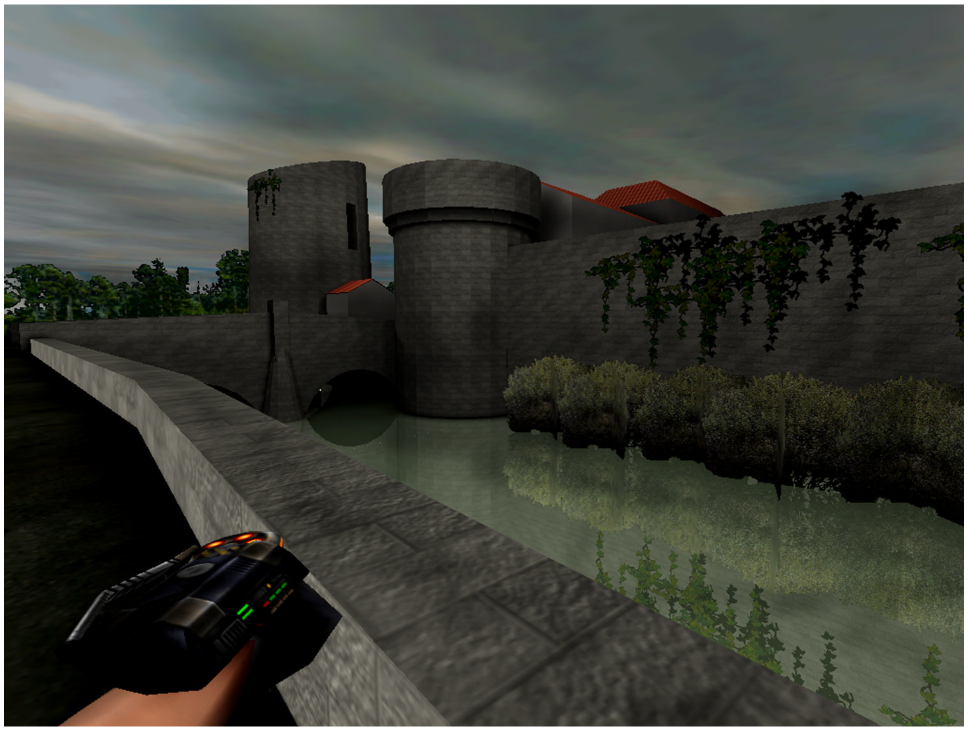


Figure 2.5 Here is the castle scene with lighting, ornamentation, and other details added.

Notice that although the textures used are base textures, they do not have to be plain in the sense that they are flat and lack punch, or depth, but they do have to tile well. Figure 2.6 shows examples of both well-done (on the right) and poorly done (the left) base floor textures from a game. Notice that a poorly done texture does not tile well, even though it is actually a texture that is simply plain concrete, than the texture with grass and cracked dirt — the image with more depth.

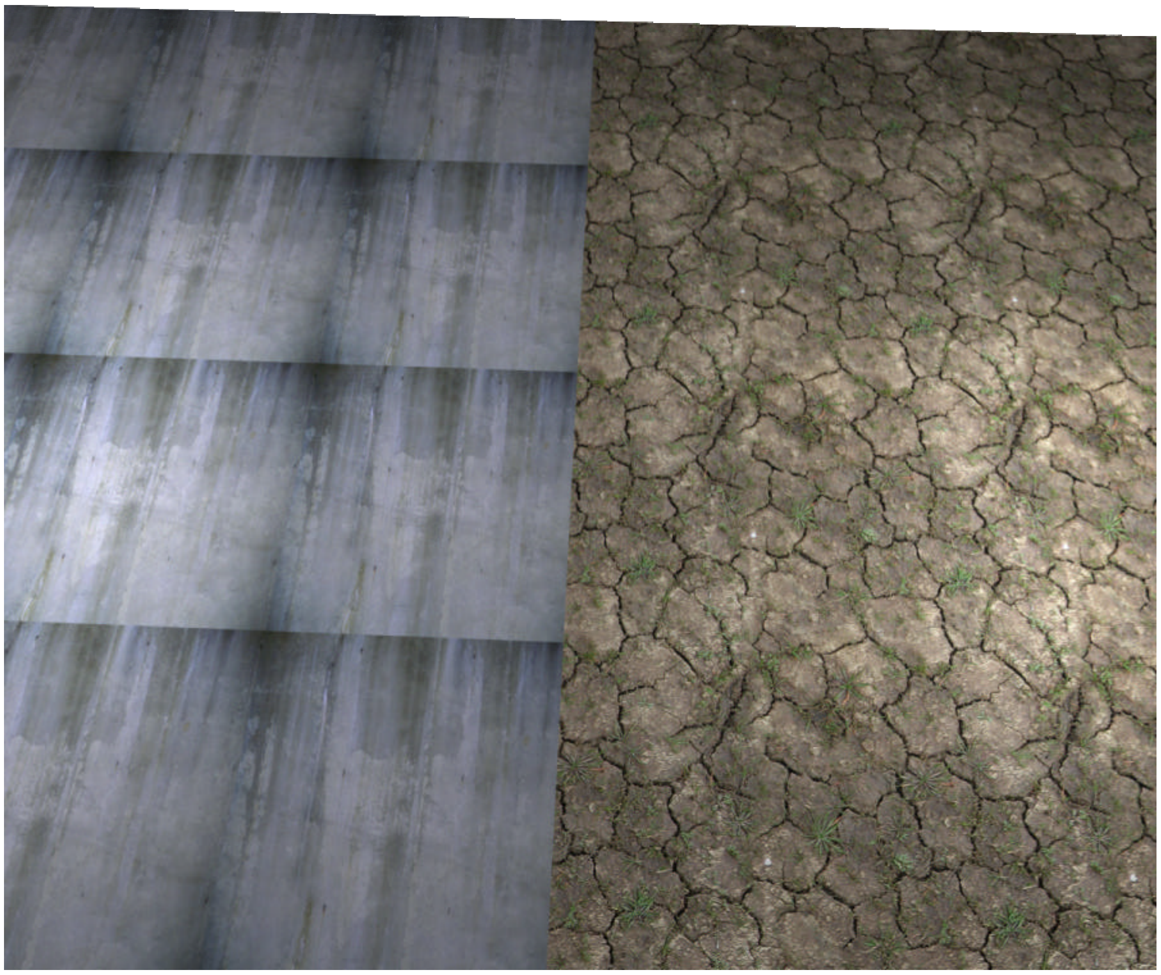


Figure 2.6 Tiling textures do not have to be plain and ugly. The plain and ugly texture to the left actually does not tile as well as the richer texture on the right.

Base textures are generally *tileable*, or laid out like the tiles on a floor. Therefore, a great deal of the texture artist’s initial job is to create textures that are a balance between tileable and deep. A *deep texture* is one that contains a feeling of depth, but the challenge is that any outstanding aspect of the texture will show up as a recognized pattern in the tiling when the texture is placed on a large game-world surface. Notice how the leaf in the grass texture of Figure 2.7 pops out when placed in a ground scene in Figure 2.8.

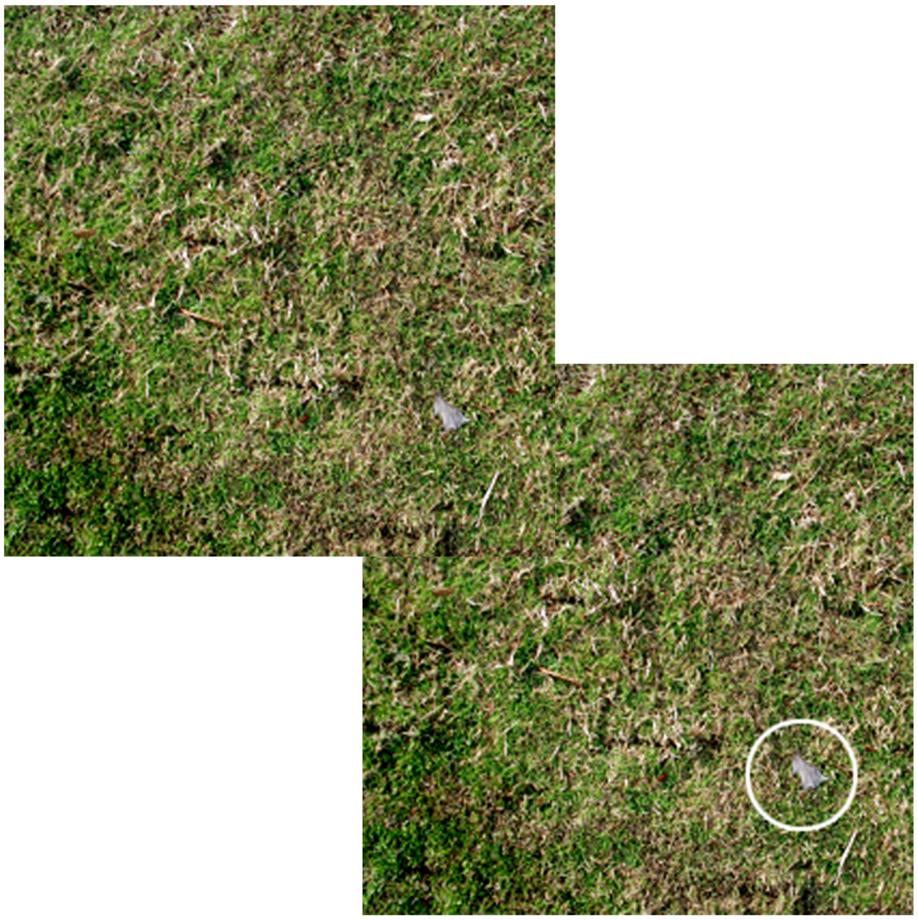


Figure 2.7 This is a great-looking patch of grass, but notice the leaf.

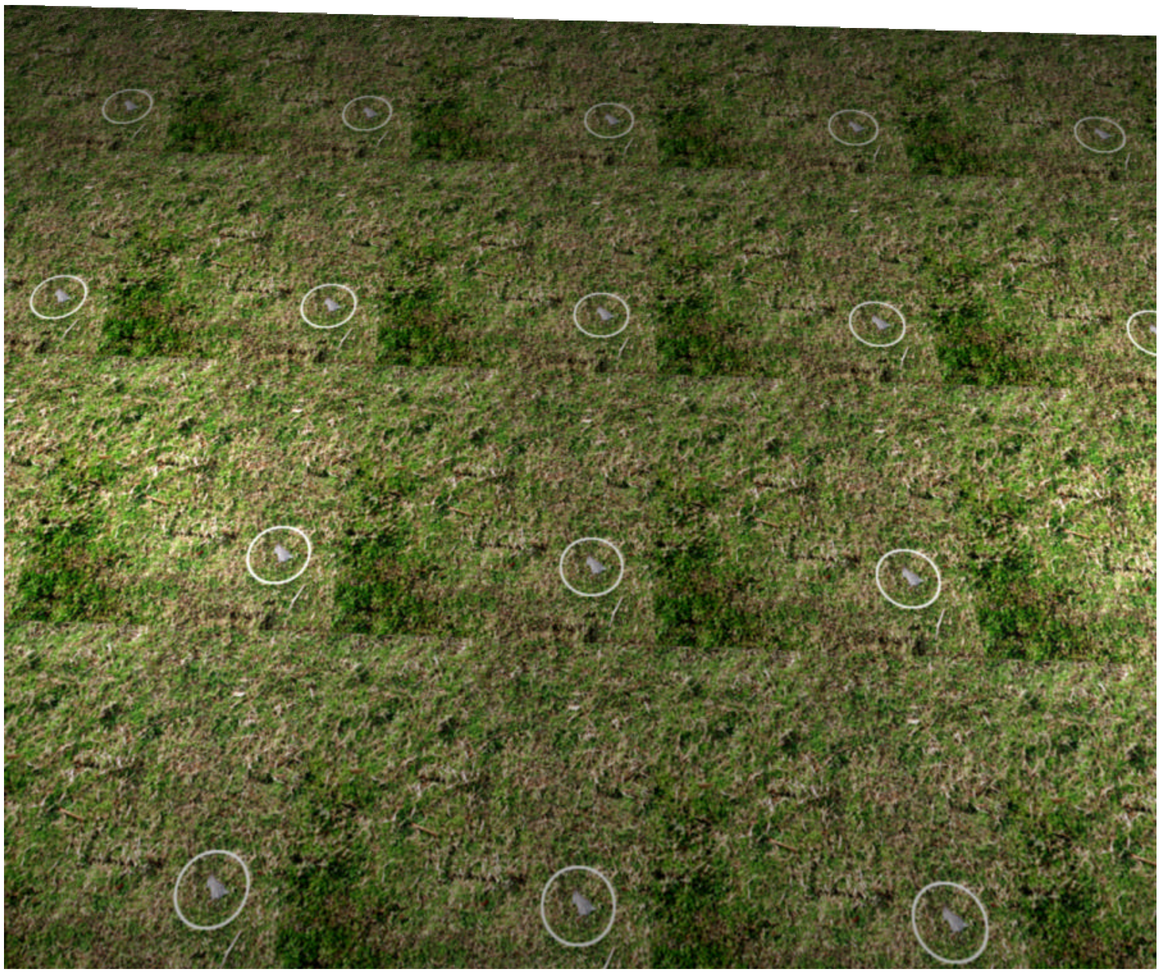


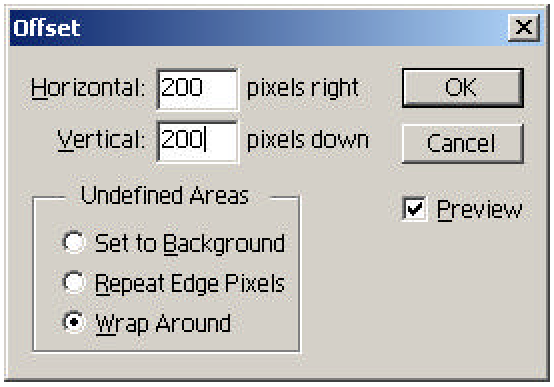
Figure 2.8 But a great-looking patch of grass may not be a great base texture; notice the leaf that draws attention to the tiling of the texture as well as that dark corner of the original image that now stands out.

Of course you can clone out the leaf in the grass texture. Cloning is a tool that you will use a great deal in creating textures, but you also have to be careful that the overall texture doesn’t appear too tiled. Only so much detail and depth can be handled in the texture itself. You will learn later how to use ornamentation, detail, lighting, and other tools to make a rich and complex level efficiently. To create basic textures, you’ll often use the Photoshop tools of filters, motion blur, noise, and other effects.

Tiling Textures

In the following exercise, you'll use Photoshop to tile an image and then to smooth the edges of the tiles where they meet:

1. Start with a new Photoshop file and make it 400 x 400 pixels. For this exercise you can use an image of your own you would like to tile or simply take one from the companion CD-ROM in the back of this book.
2. Now choose Filter | Other | Offset (see Figure 2.9), and enter the following values: Horizontal 200 and Vertical 200. The Horizontal and Vertical values are the number of pixels to offset the image, so our 400-pixel image offset at a value of 200 pixels will move the image halfway across the canvas. Make sure you have Wrap Around checked and Preview if you would like to see the texture updated as you work. Wrap Around tells Photoshop to repeat the image and not simply move it over by the pixel value you enter.


Offset

Horizontal: 200 pixels right

Vertical: 200 pixels down

Undefined Areas

( ) Set to Background

( ) Repeat Edge Pixels

(•) Wrap Around

[✓] Preview

Figure 2.9 This is the Offset dialog box, which you will see a great deal as a game artist using Photoshop.

You will notice that what happens to the image is that the lower-right quadrant is put onto the top-left quadrant of the image, and so on. In effect it looks as if four copies of the image are placed together and then the center of the four images is placed on the edges of the new image. See Figures 2.10 and 2.11 for an illustration of this concept.

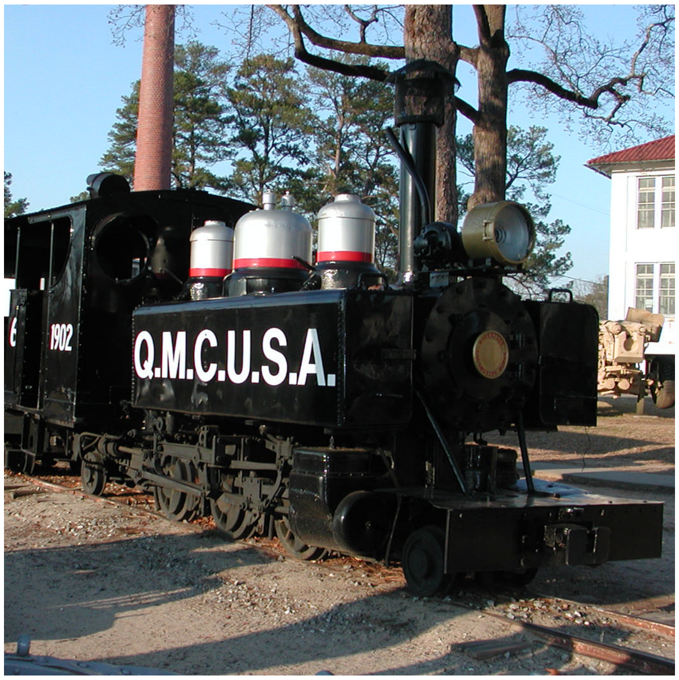


Figure 2.10 This is the original image before we tiled by using the Offset filter.

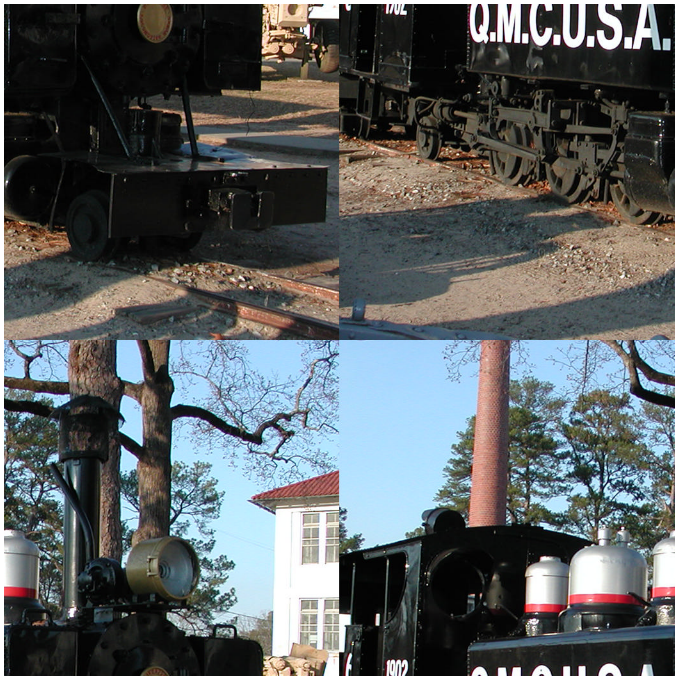


Figure 2.11 This is the image after the Offset filter was used on it. Notice the rearrangement of the image.

3. Now you should see some harsh edges or seams (see Figure 2.12), and this is

where the Clone tool comes into play. Using the Clone tool, you can blend the edges of the image where the clouds meet. Usually you will want to use a larger, softer brush and clone near the area you are cleaning up.

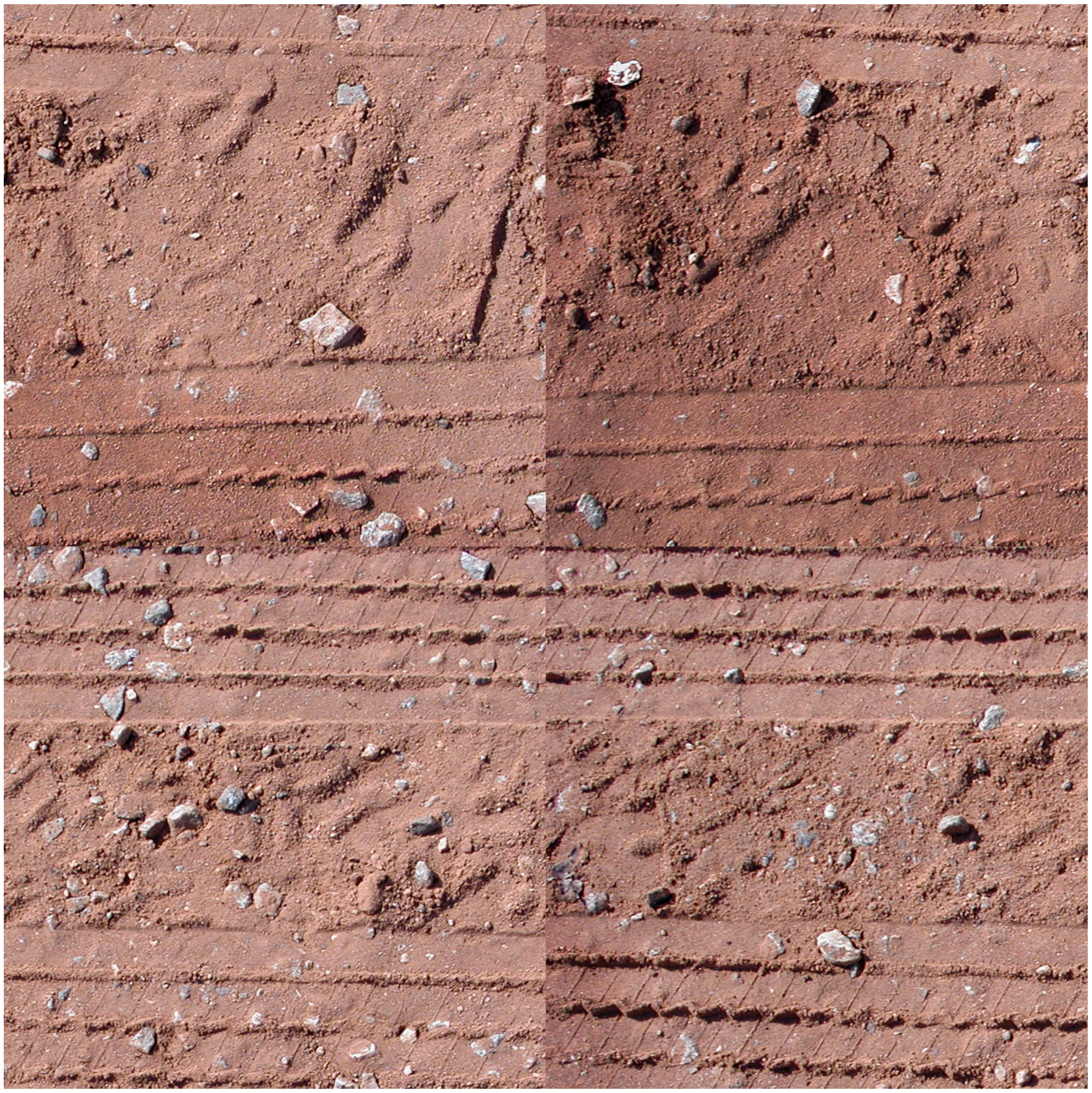


Figure 2.12 Notice the clearly visible seams where the offset images meet.

Troublesome Tiles

Sometimes you may be trying to tile an image that is unevenly lit or in some way a troublesome image when you're trying to use the Offset filter method of tiling. In these instances you can do the following, or some variation, depending on the parts of the image that are troublesome to your tiling:

1. Open the image named ‘window’ from the companion CD-ROM. Use the Square Selection tool, and select and copy the half (or even quarter, in some cases) of the image that looks best (see Figure 2.13).

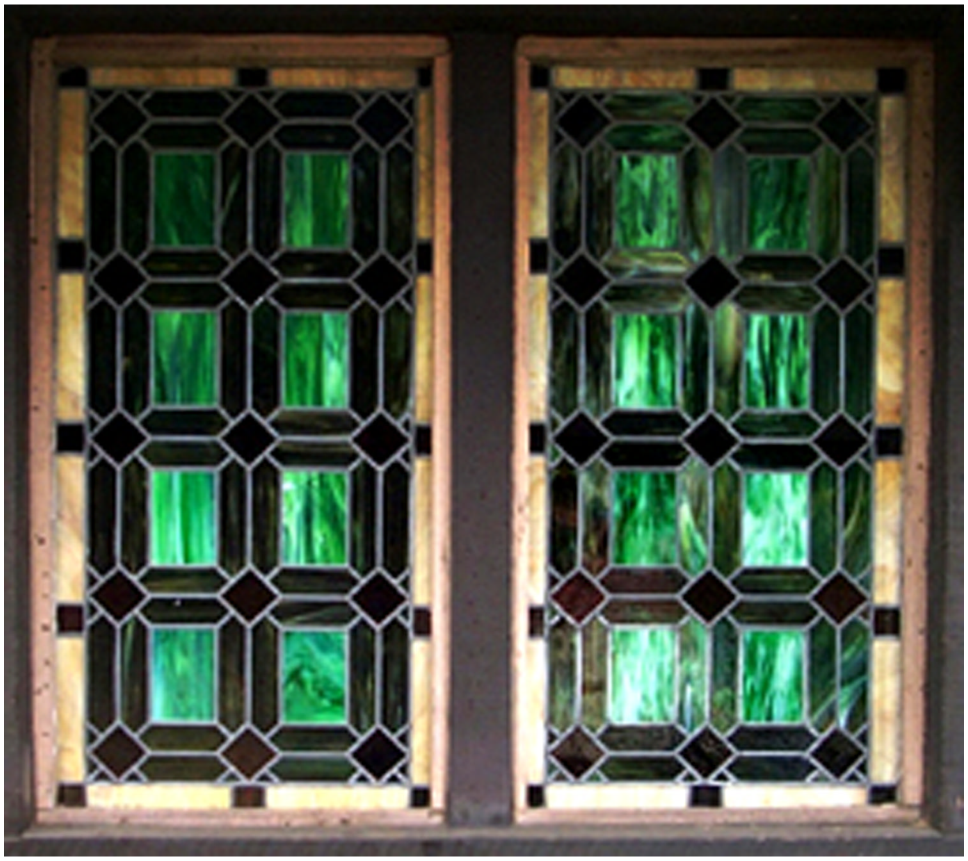


Figure 2.13 Notice that the bottom half of the window is lit by the sun and the top half is in shadow.

2. Paste the copied portion onto its own layer.
3. Depending on the orientation and problem area of the image, you will now flip the copied portion of the image. Using the window image we are working on, we will copy the bottom half of the image and then Vertically flip it so the center lines up perfectly, and the top and bottom edges are actually the same (see Figure 2.14).

   Note: Do not rotate the image; flip it. If you rotate it, the image will not line up properly.

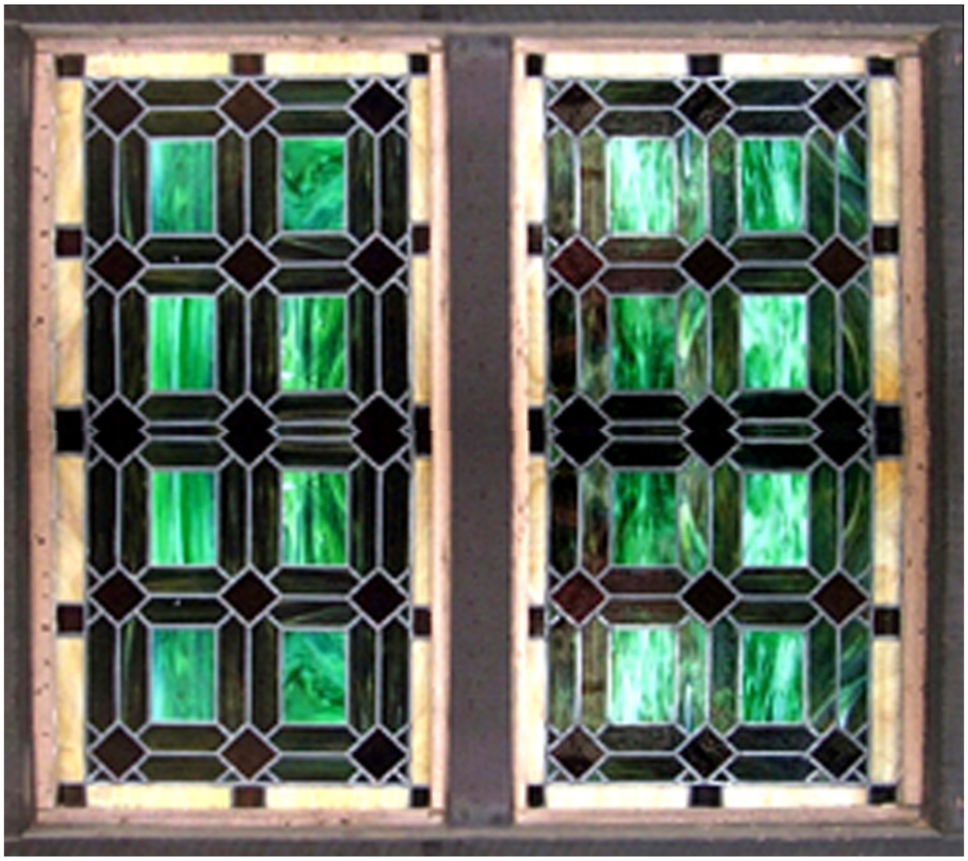


Figure 2.14 See how the bottom half flipped creates a perfect copy of the bottom, only now on the top for good tiling (also notice the hard seam).

4. There may be a seam where the layer with the copied portion of the image lies over the old version. If so, you can erase the hard seam with a soft brush. This usually works surprisingly well. See Figure 2.15.

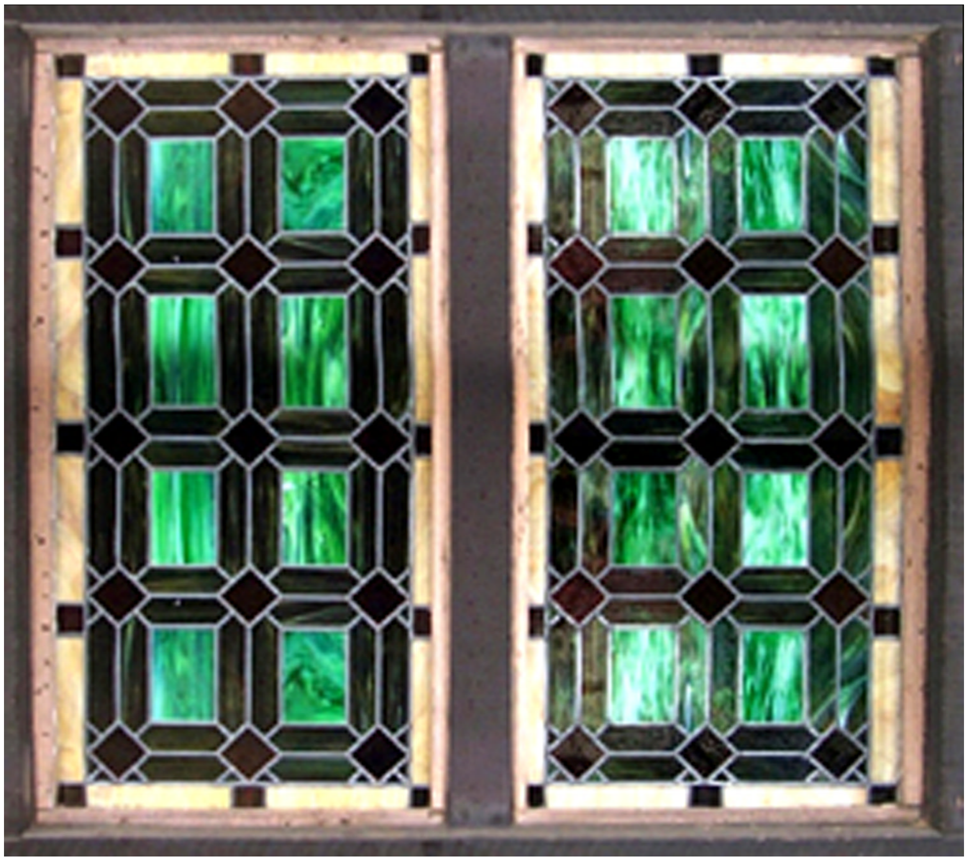


Figure 2.15 Erasing the seam with a soft brush creates a seamless transition between images.

Testing The Image Tile

When you are finished cloning the image and smoothing it out, you can test how it tiles in a few easy steps. You can offset the image again to get a general idea of how well it tiles (pressing Ctrl+F will reapply the last filter you used). Or, for a much better idea of how well the image tiles when applied repeatedly over a large surface, you can do the following:

1. Flatten the image (a command in Photoshop that combines all layers in the image into a single layer). Select the entire image by choosing Select | All or pressing Ctrl+A. Then choose Edit | Define Pattern. This saves a copy of the selected pattern in RAM.
2. Open a new image. If your system can handle it, make the image exactly twice the size of the original, or more in multiples of 2 from your original image size. In this case, an 800 x 800 image will work.
3. Now choose Edit | Fill and select Pattern. You will see the image tiled across the new canvas, and you will see if there are any seams or hot spots that draw your eye (see Figure 2.16).

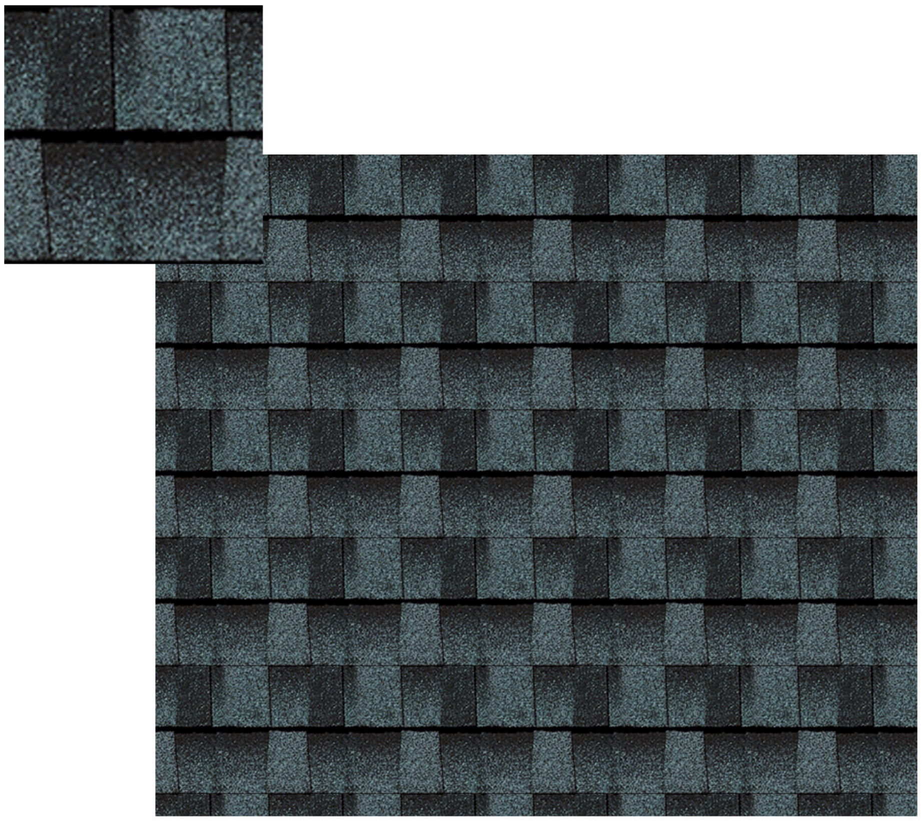


Figure 2.16 To see if an image tiles well, we're testing it by repeating it over a large canvas.

Tips for Horizontal and Vertical Tiling

Sometimes textures are supposed to tile in only one direction, either vertically or horizontally. You'll tile vertically in the case of square support columns, for example, where there is only a seam on the top and the bottom of the image that needs to meet up. You'll tile horizontally in the case of a wall texture where there is only a seam on the left and the right of the image to tile. See Figures 2.17 and 2.18 for examples of horizontal and vertical tiling.

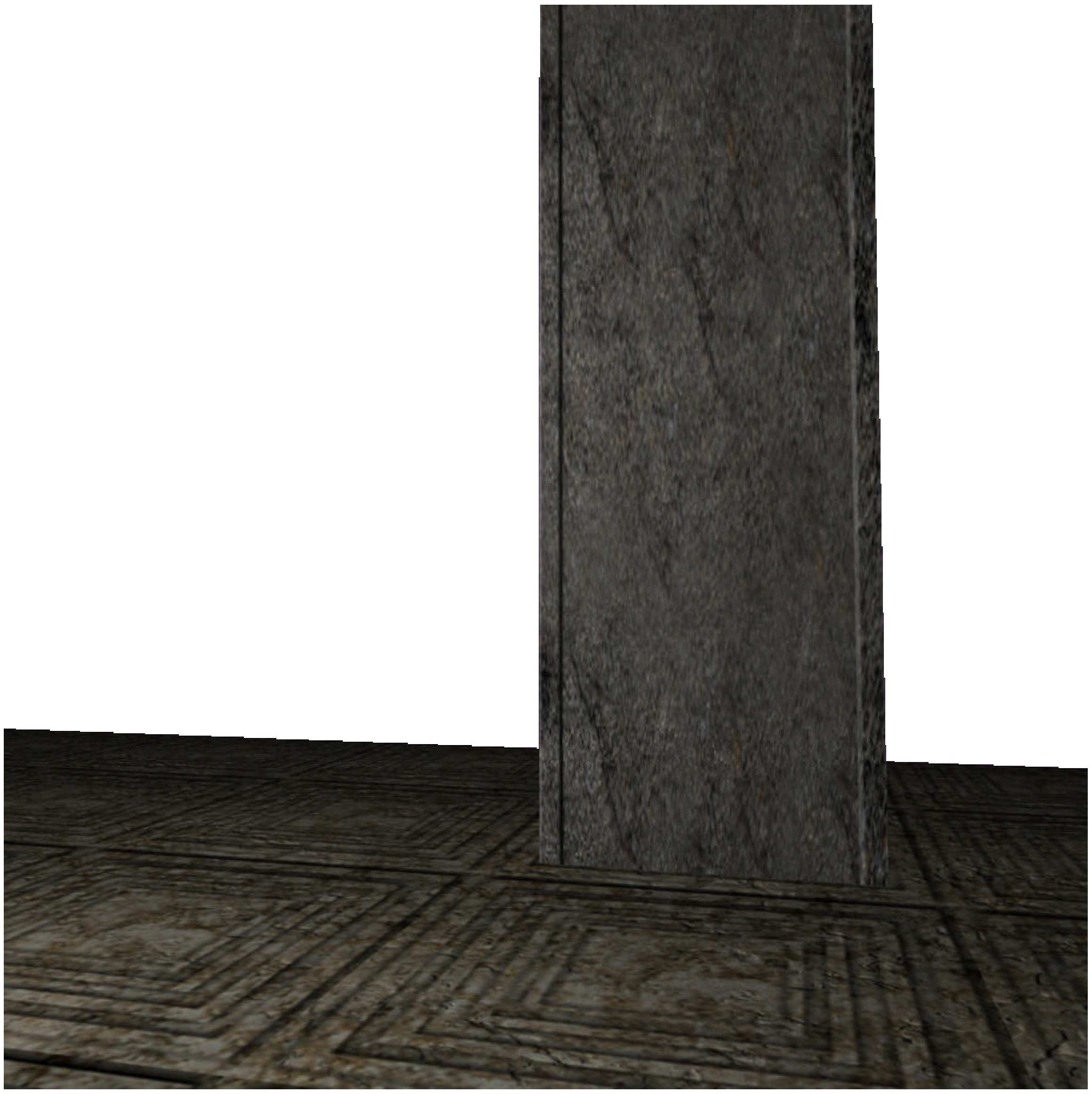


Figure 2.17 See the vertical tiling on this column. If this tile were tiled horizontally, it would not look quite right.

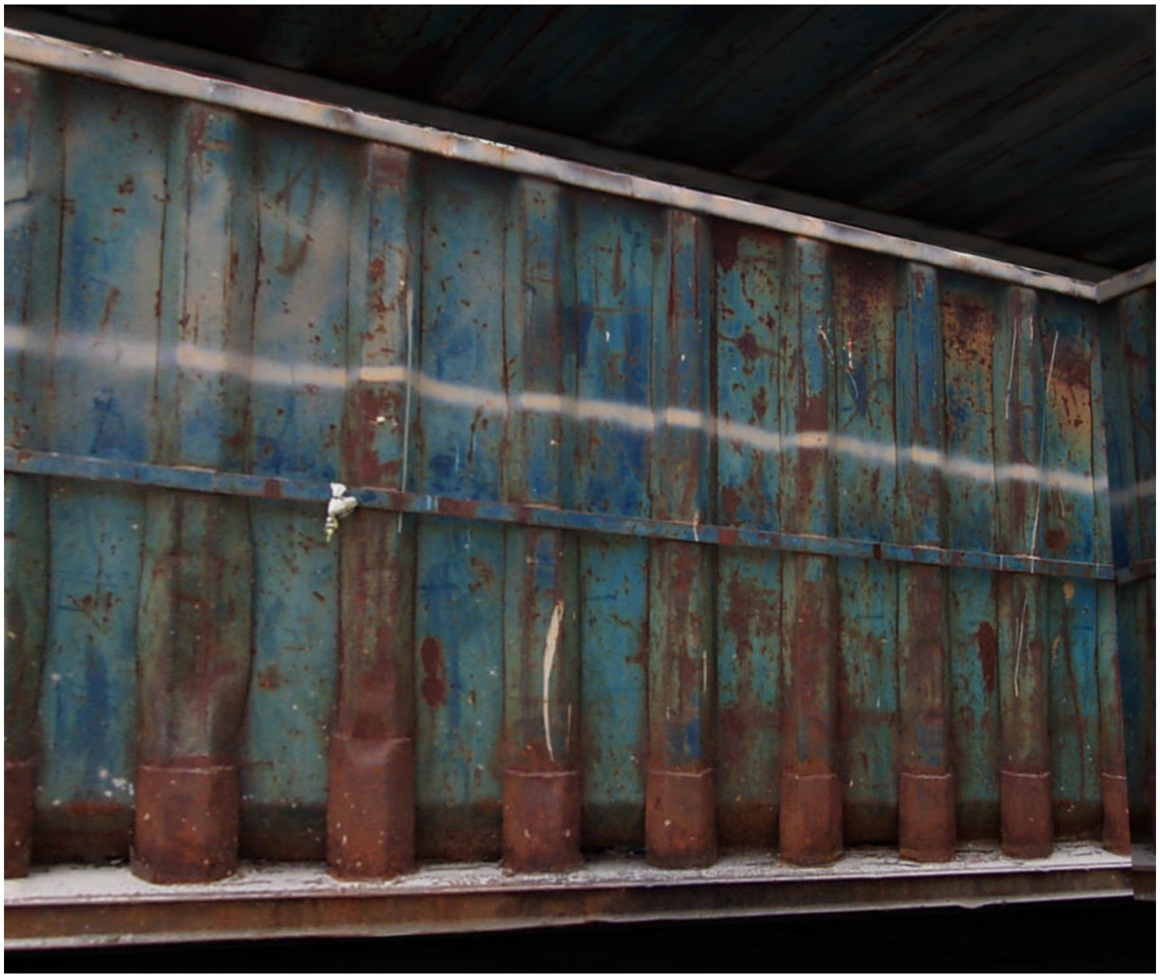


Figure 2.18 This tiled wall looks good horizontally, but if it were tiled vertically, the dark bottom and bright top would clash.

These one-way tiling textures are usually a bit easier to create because you are blending only one seam as opposed to four. You may notice that when you're trying to blend a seamless tile that tiles in both the horizontal and vertical directions, you also have to deal with the place where the four corners meet as well as the seams. This middle spot is usually more difficult to clone and blend than a straight seam is. Not only is the blending easier on one-way tiles, but you can also cut and paste to make flawless tiling easier. For example, see Figure 2.19.

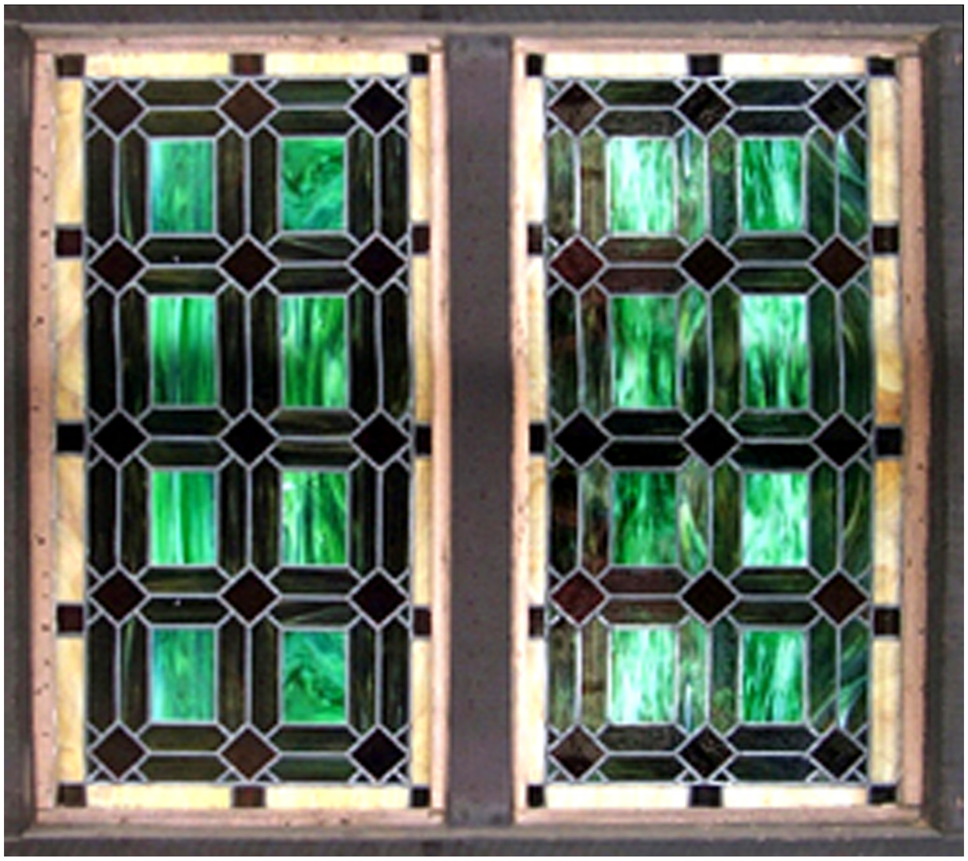


Figure 2.19 As in the previous example, copying and flipping half the window makes the tiling flawless.

Of course, when you’re tiling a texture that has seams in the image already--such as items from the real world such as paneled walls, floor tiles, or repeating patterns (see Figures 2.20 through 2.23)--it is always best to use those seams to your advantage when tiling the texture.

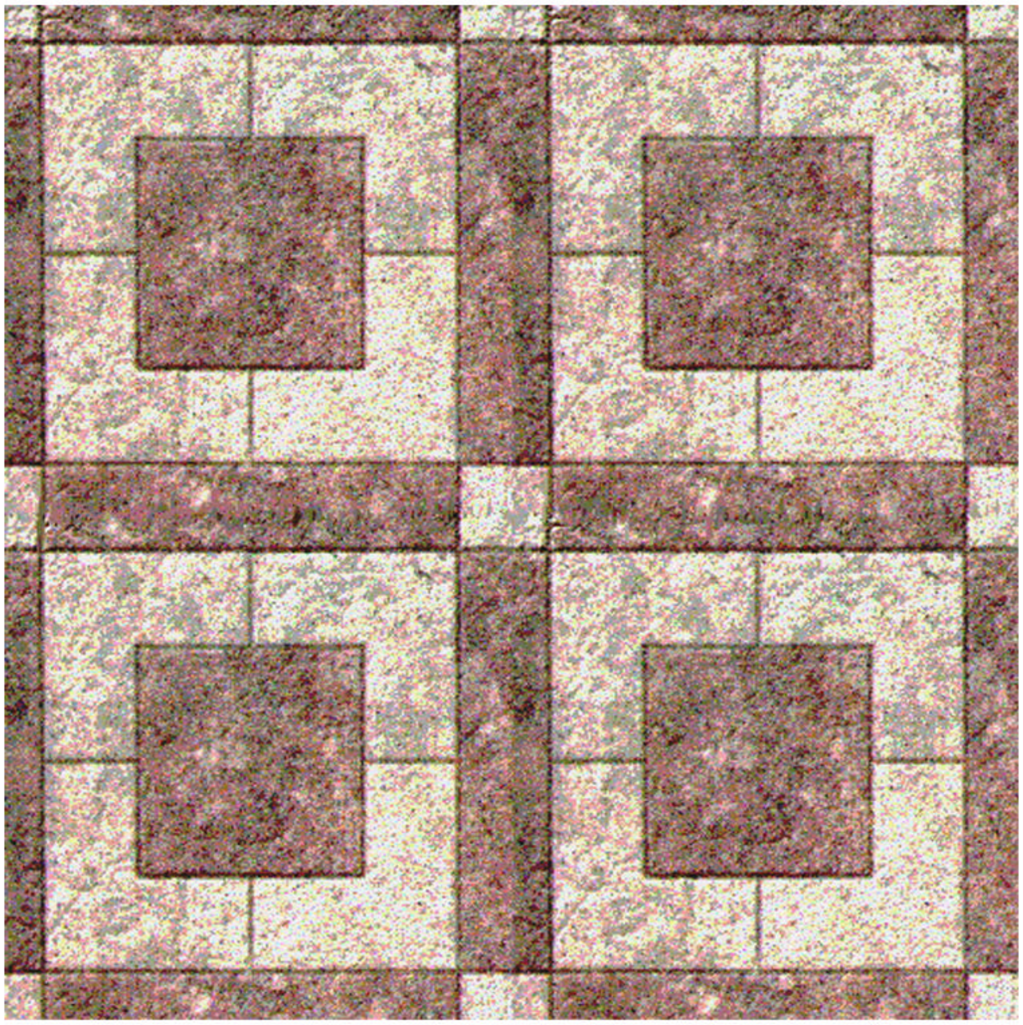


Figure 2.20 A floor tile that is supposed to be a tile is a prime example of using seams to your advantage.

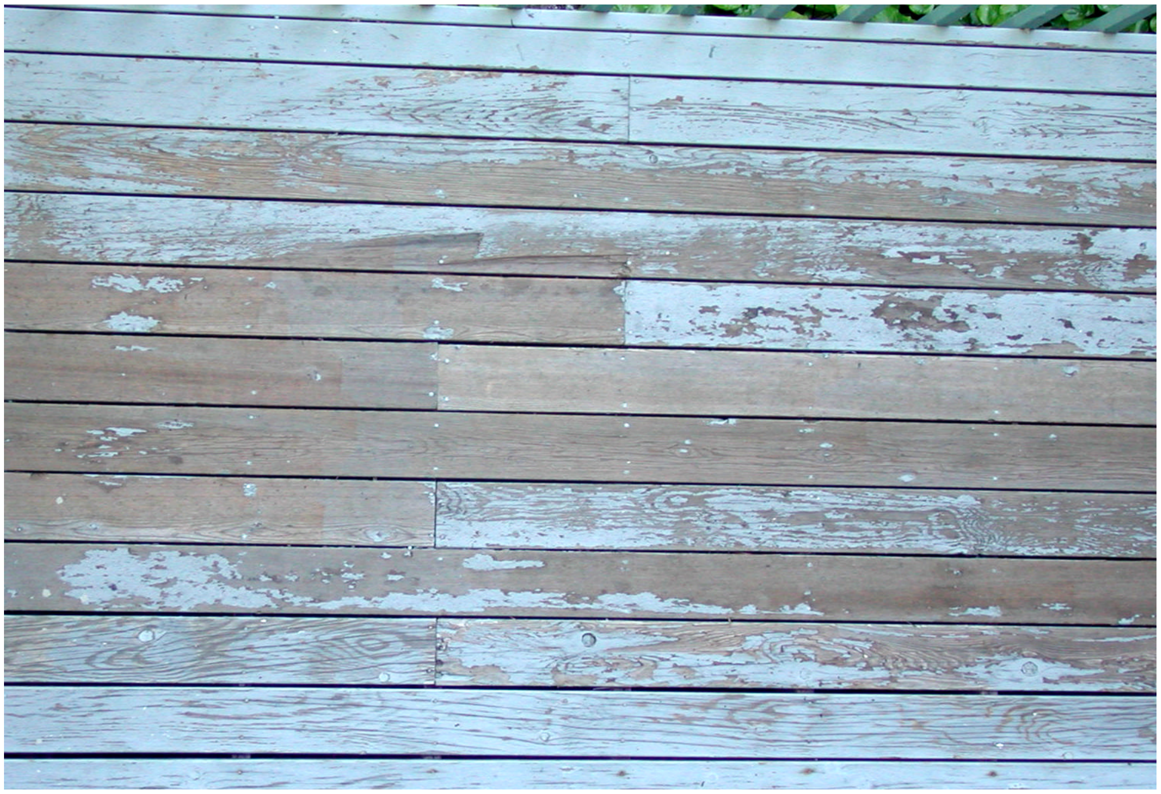


Figure 2.21 This tiling plank wall has many seams and gaps that can be used to aid the texture artist in creating a cleanly tiling image.

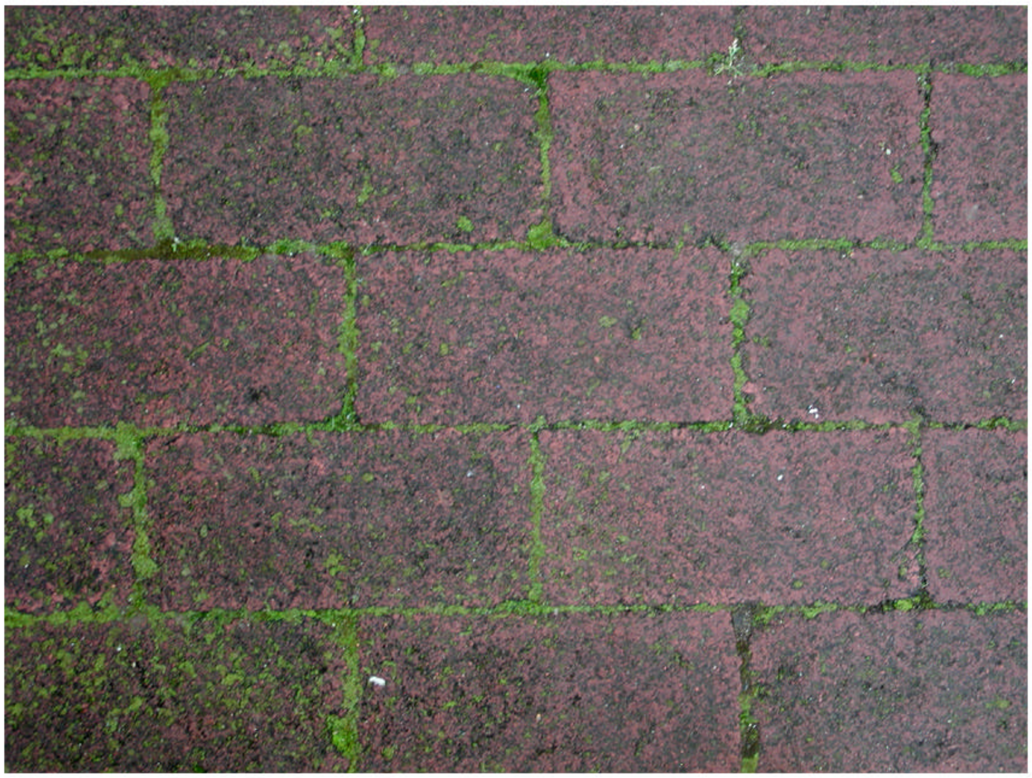


Figure 2.22 Even stones have many seams and spaces that allow for easy tiling.

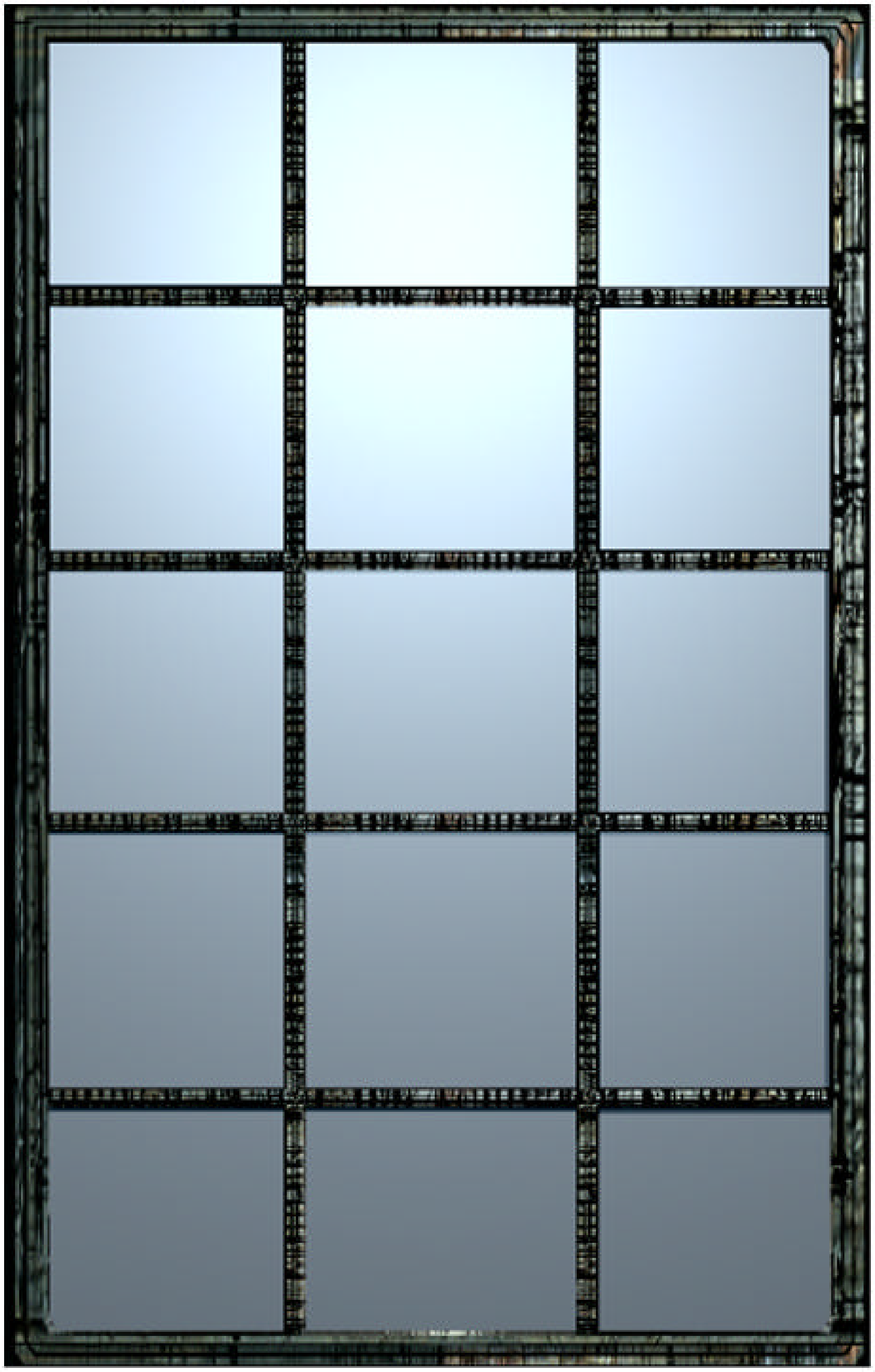


Figure 2.23 Windows are among the easiest things to tile. They can be tiled by the pane as well and used to cover small and large holes with the same texture.

Advanced Tiling: Natural Images

Tiling some images is easy, while others may present quite a challenge. Some of the hardest to tile are the natural images from digital photographs. For example, we will use an image of some cobblestones in the next exercise. Achieving a smooth and continuous tile with an organic pattern such as these cobblestones, as opposed to cinder blocks or some symmetrical pattern, is a challenge.

Tiling Cobblestones

This exercise involves tiling an image of cobblestones in Photoshop:

1. This exercise is much easier with the grid on, so go into the Preferences and set the grid size to any number as long as there are at least 8 to 10 grid squares over your image. For the image in this exercise, set the gridlines to every 32 pixels.
2. Now open the image named stone.jpg from the folder for this chapter (see Figure 2.24).
3. You will notice that the image is 640x480 pixels, but we eventually want a 256x256-sized (square) image. So we start by making the image square. We are first faced with the decision of how we do this. Do we stretch or distort the image? No, we have to choose the portion we will crop out. I chose the middle of the image where it looks the cleanest. Now if we use the Crop tool to outline an 8x8 square area, we will have our 256x256 image, but we want to outline a 10x10 square area so we have the border needed to make the tile work right (see Figure 2.25).

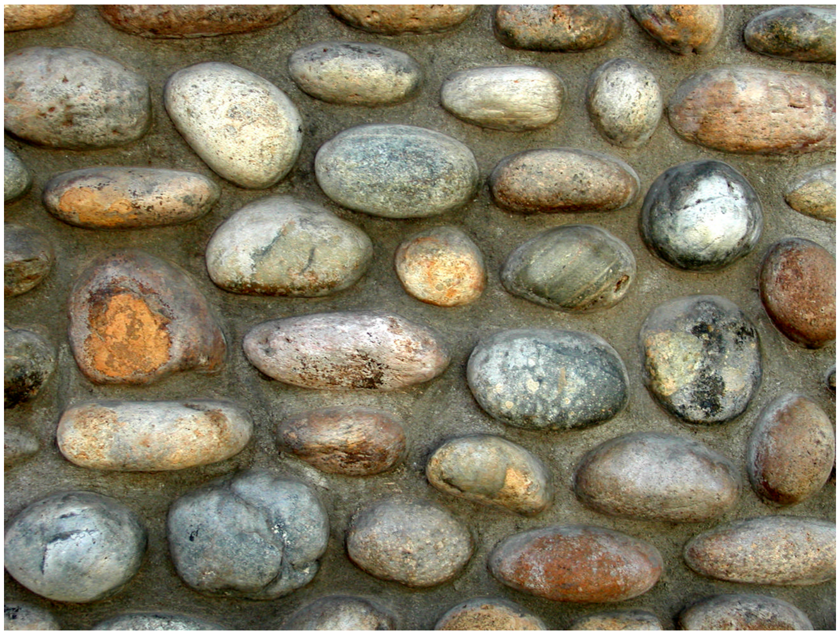


Figure 2.24 This is the raw cobblestone image we will be tiling. Tiling organic patterns such

as this can be very challenging.

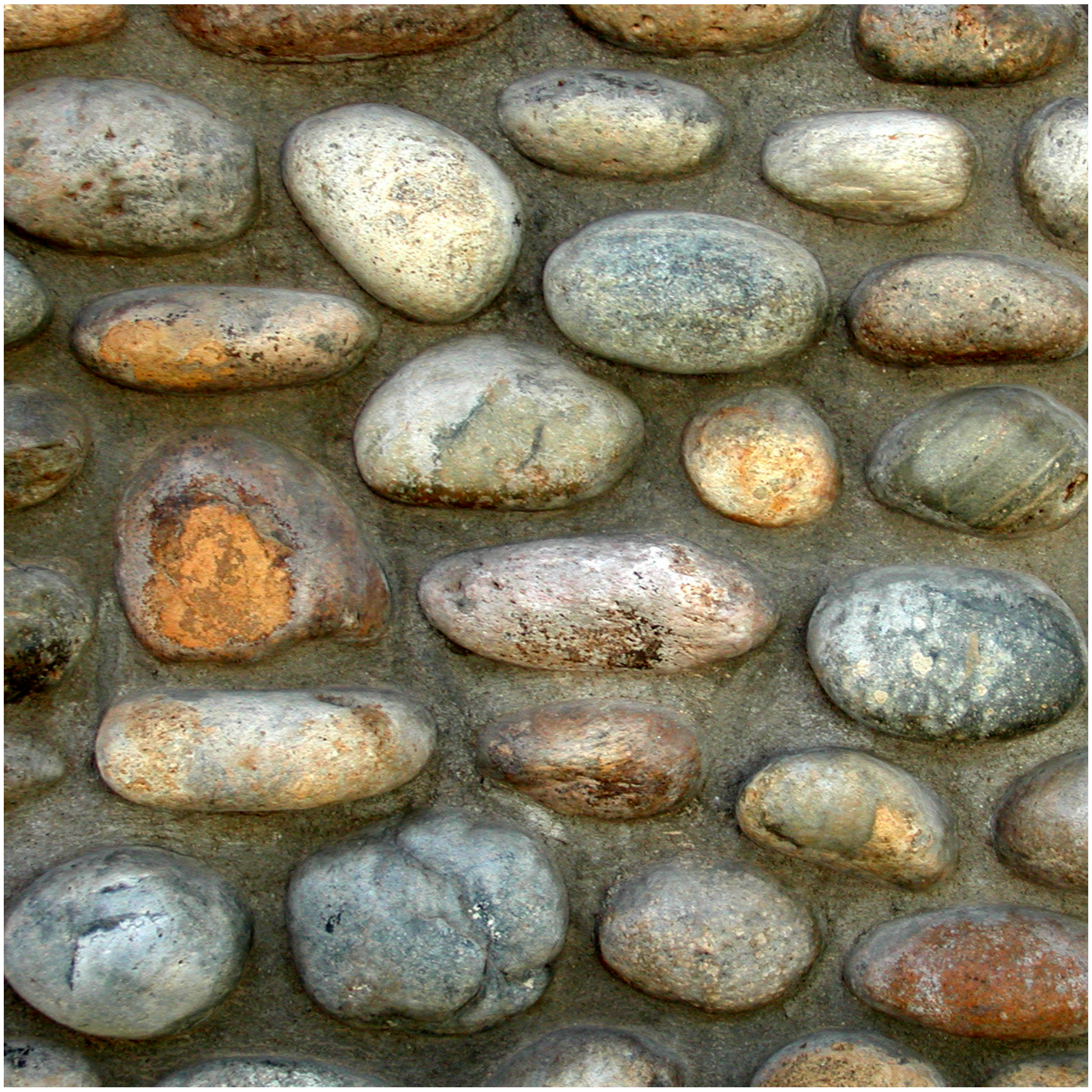


Figure 2.25 Notice how the image has been cropped to make it square, also keeping in mind that we need to remove any elements that may make tiling difficult.

4. Select the bottom two grid-rows of the image. Copy this selection by pressing Ctrl+C, and then paste the copy by pressing Ctrl+V because this will automatically create a new layer. Hold the Ctrl key and click and drag the selection to the top (see Figure 2.26). Having the grid lines visible and the Snap option turned on will make this step easier. Use the Eraser tool to remove the area from the small selection at the top and make the seams seamless. Use a medium-sized, very soft brush and work carefully.

Note: Try clicking the Eraser tool off and on as you work as opposed to holding it down continuously. When you make a mistake while erasing and you use the Undo function (Ctrl+Z), it will undo all the work done while the tool was active.

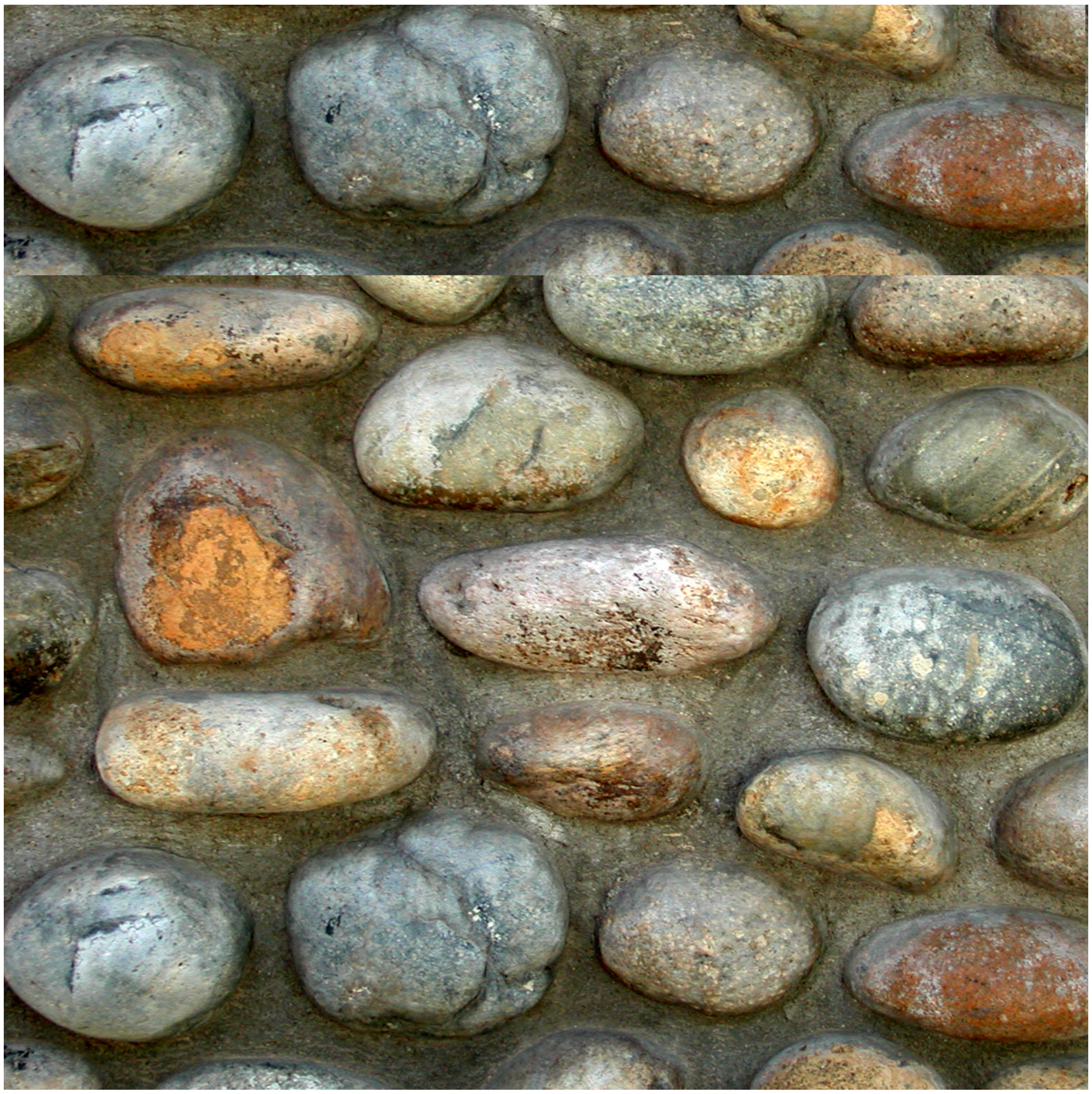


Figure 2.26 Here we copied the bottom two grid-rows of the image and pasted them on the top of the image.

The reason we use this small selection--rather than copying half the image, flipping it, and then erasing the middle of the image--is that this helps reduce the tiling you might see if a larger portion of the image were repeated using the previous method. The previous method works well where you have smoother surfaces with lighting you want to normalize.

5. Now you can use the Crop tool. Select everything but the bottom rows you copied, and crop them from the image, and you will have an image that seamlessly tiles, or at least vertically it does right now. Now we can tackle the horizontal tiling of the image. Start by flattening the image and repeating the process from above. Take the two grid-rows from one side and drag them over to the other. Clean, crop, and you are in business.
6. Remember, when you're tiling a texture in both the vertical and horizontal directions, you have to be careful that the corners are clean. To achieve this, work on the middle 85-90% of the seam and leave the corners for another step. When you have most of the middle section of the texture cleaned up, duplicate the layer and use the Offset filter on the top layer. Do your cleanup on the top layer where the corners meet.

Changing Lighting

Now that you are the master of tiling textures, let's look at some final tips for making your textures their best. This stone image we used looks great on a cobblestone street during the day, but if you were to use this in the streets of a game world where it is supposed to be night time, it would look funny – washed out and too bright. Instead of simply dragging the Contrast and Brightness settings up and down, we can make the stone look like as if it were really on a night street in some dark and scary fantasy city.

In Photoshop, follow these steps:

1. Load the tiled stone image you've just created.
2. Select the entire image (Ctrl A), copy it (Ctrl C), and then paste it (Ctrl V). You now have a new layer with the duplicate image.
3. Run the Gaussian Blur filter on this new layer (choose Filter - Blur|Gaussian Blur) and make it fairly blurry; see Figure 2.27.

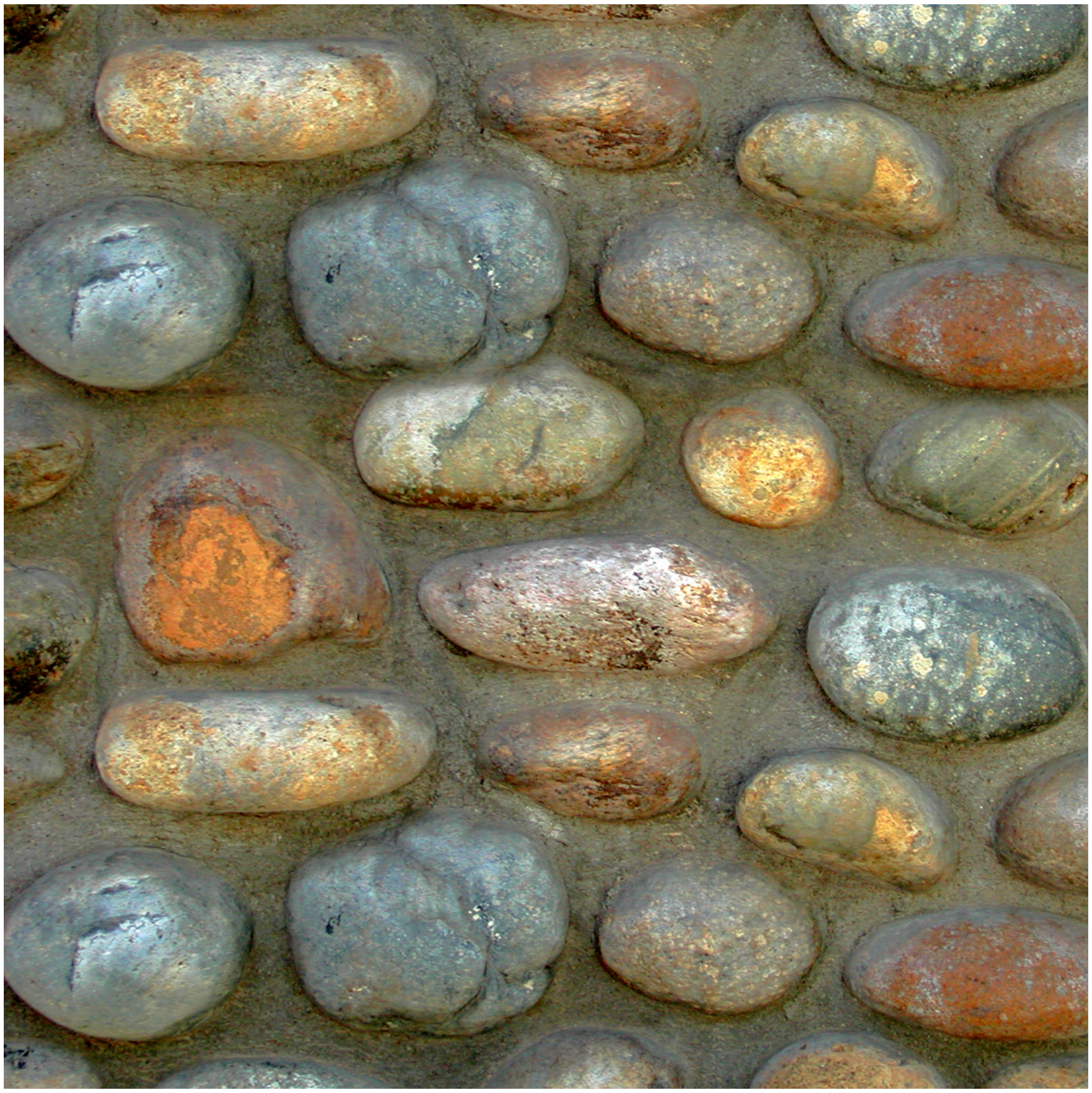


Figure 2.27 The copied layer with the Gaussian Blur filter applied.

4. Now invert the image (Ctrl+I) and desaturate it (Shift+Ctrl+U).
5. Set the layer mode to Overlay.
6. Now it is a simple matter of adjusting this layer's brightness and contrast (choose Image|Adjust|Brightness/Contrast) to make the image look like you want it to. Figure 2.28 shows the original version and the finished version. Figure 2.29 shows them both used in a game scene that is supposed to be at night.

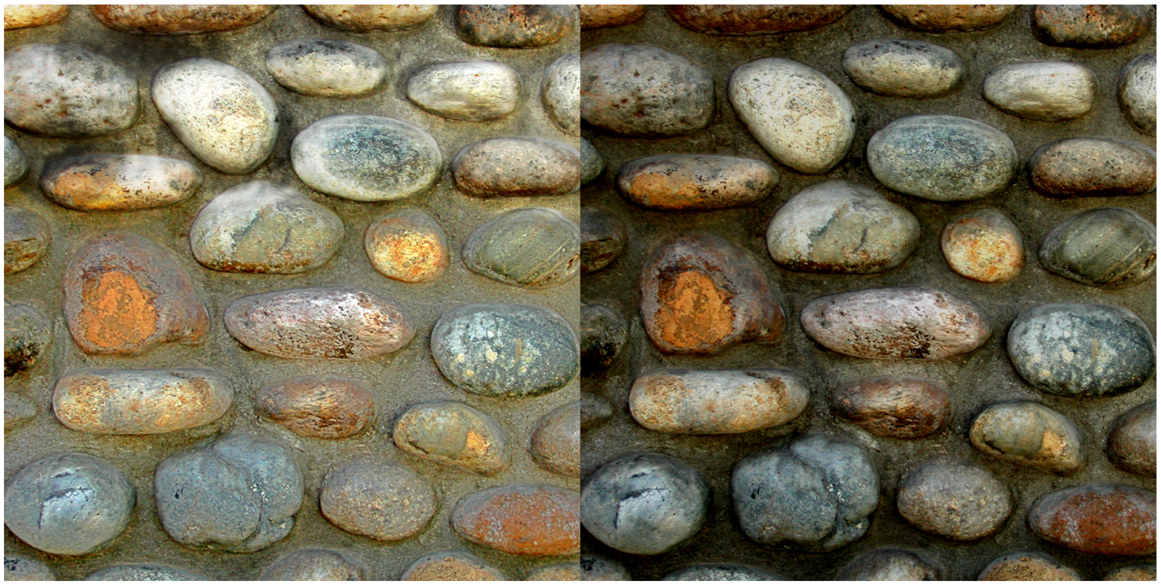


Figure 2.28 Here is the first (original) and the relit versions of the cobblestone images side by side.

Figure 2.29 Here both cobblestone images are used in the same scene in a game world: day (left) and night (right).

Creating Base Textures

Now we will look at various ways of creating textures. Which is the best way? There really is no one best way. You will use all the tips and tricks presented here, depending on your needs, and as you develop more textures and games, you will undoubtedly learn many more. Whether you start with scanned-in art or digital stills, or you are creating all your textures in Photoshop, you will always use a bit of each technique to get your textures just right.

The easiest and quickest way to create base textures is to use Adobe's Photoshop or similar such image manipulation program. Photoshop allows you to create game textures completely within the package, or by using other assets as a base. To start with, we will look at creating textures from the ground up using Photoshop. Knowing how to do this will allow you to quickly generate placeholder assets for your project so the programmers and level editors have assets to work with as you perfect your textures. And you will never be stuck looking for a good patch of stone to photograph or waiting for the programmer to bring the digital camera back to the office.

In the following exercises, you'll use Photoshop to create a few commonly needed assets in games. These assets can then be used to create more detailed and complex textures with techniques you will learn a little later.

Creating a Rust Texture

Every game has a need for rusty metal, whether you are creating a space port with rusty walls and control panels, or a medieval fantasy world with rusty metal hinges and weapons. Rust is the game artist's best friend.

To create a rust texture in Photoshop, follow these steps:

1. Create a new image document and make it 600 x 600. I deviate from the larger 800 x 800 size sometimes because the filters in Photoshop are affected by the size of the image.
2. Fill the background with a very light brown. I used RGB 158, 139, 117.
3. Add noise to this layer with the Noise filter: Choose Noise | Add Noise. Set the amount to 40, Gaussian, and make it Monochromatic.
4. Blur this layer by choosing Blur | Motion Blur. Set the angle to 45 and the Distance to 45. Your image should look like Figure 2.30 by now.

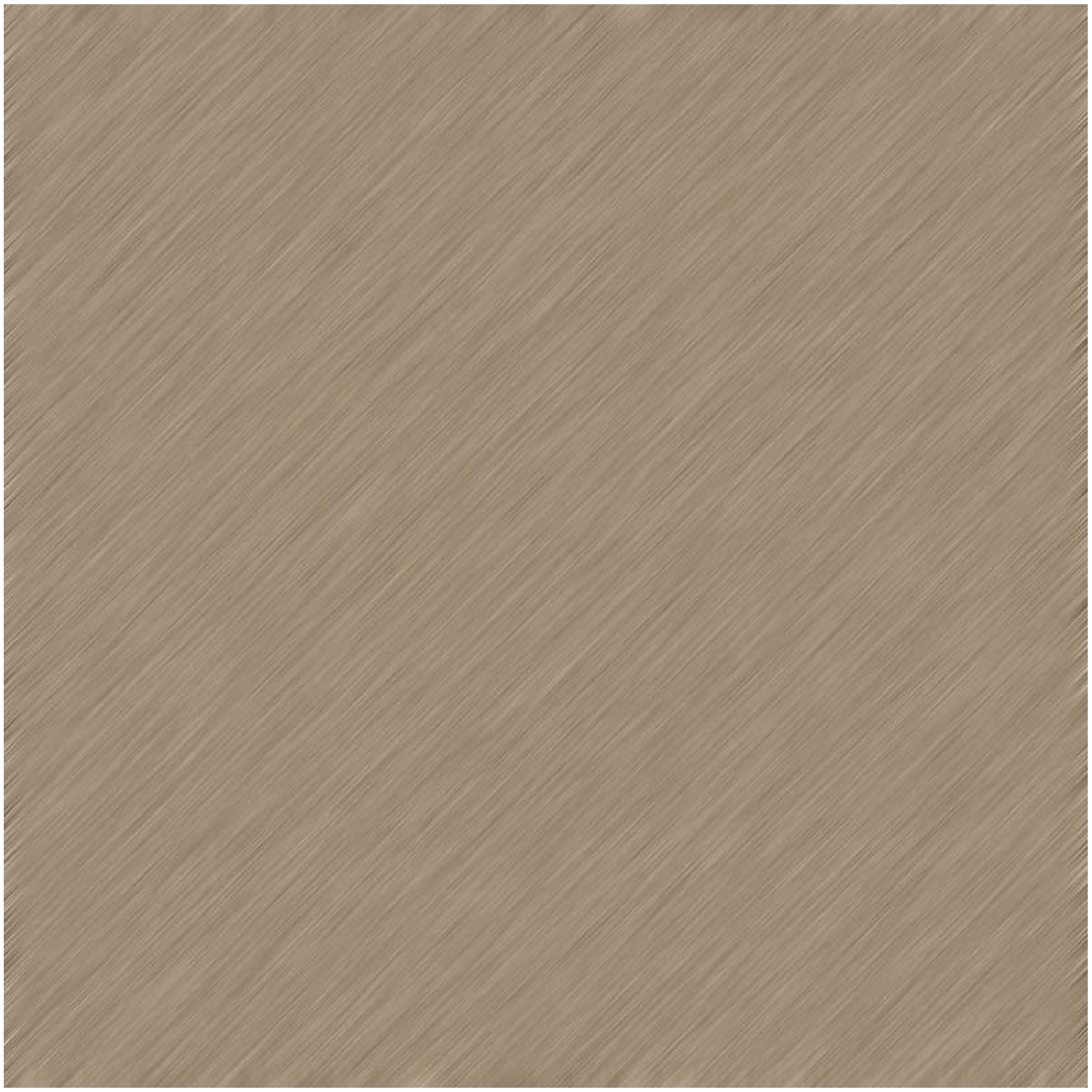


Figure 2.30 Here we see the beginning of a rust texture.

5. Distort the layer by choosing Distort | Ocean Ripple. Make the Ripple Size 9 and the Magnitude 9.
6. Make a new layer, and select a darker shade of the brown we used earlier (I used RGB 104, 66, 24).
7. Now use your Paintbrush, and with a really soft brush, paint some random lines back and forth (see Figure 2.31). You may want to Motion Blur this layer a bit if you don’t like how it looks when we are done.

Figure 2.31 A paintbrush, some random lines, and the Motion Blur filter will spice up this image.

8. Change this layer's mode to Color Burn and set the opacity down a little.
9. Create yet another layer, and set the background color to black. Render some difference clouds (choose Filter|Render|Difference Clouds), change the mode to Color Burn, and turn down the layer opacity to about half. Difference Clouds generate a cloud pattern like Render Clouds, but instead of filling the image layer with the clouds it blends the generated Clouds with the existing image.
10. Create another layer. Fill the layer with the brown color again and add some noise. Change the layer mode to Soft Light.

Now you have a really nice base rust texture (see Figure 2.32). We will be using this texture later in several exercises to make walls and other metal objects.

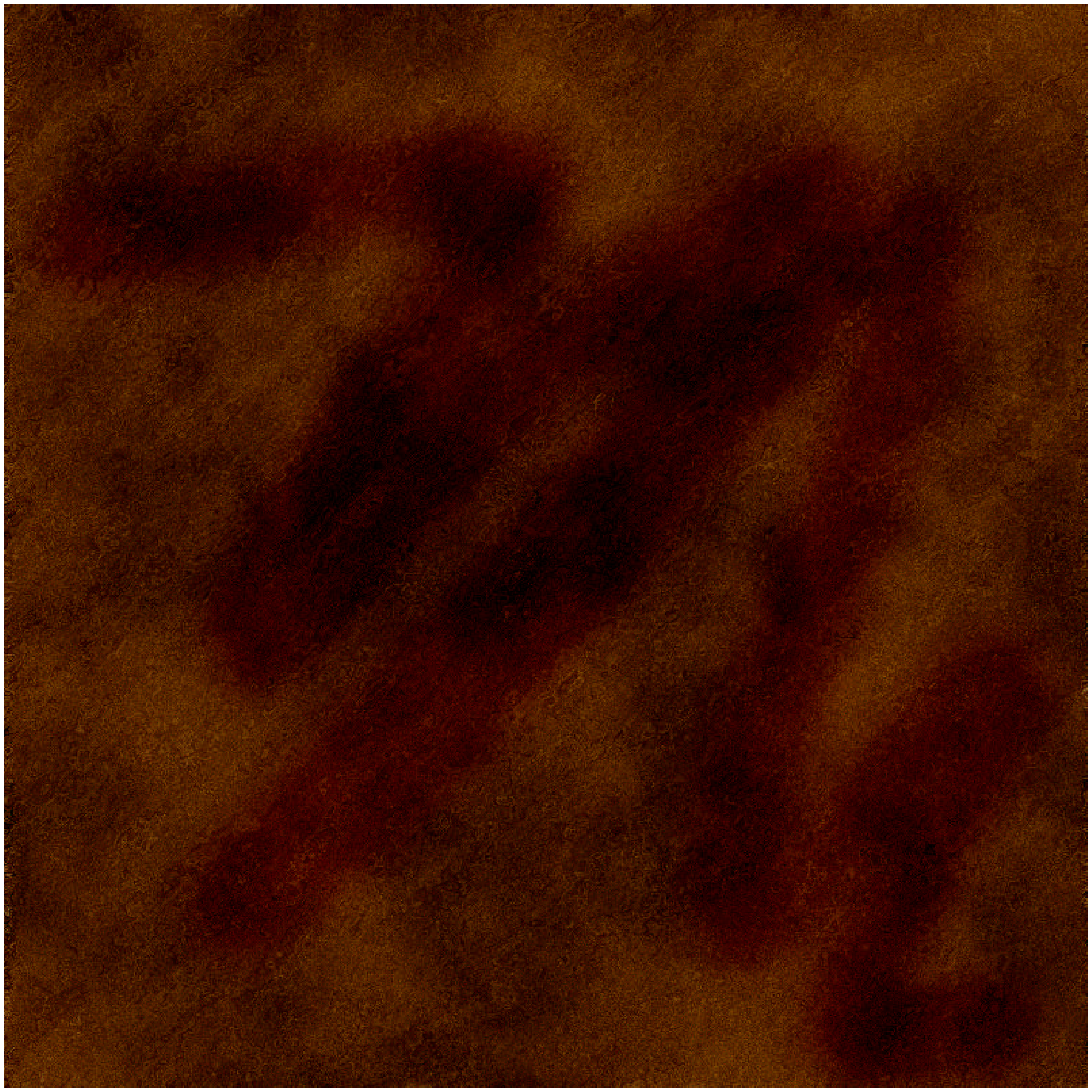


Figure 2.32 A really nice base texture for rust that took only minutes.

Creating Brushed Metal

Brushed metal is easy to create in Photoshop and is very useful as a base. As always, you work from large to small, so let’s start by creating a new image that is 800 x 800 pixels:

1. Start with an 800x800 image. Set the background color to black and the foreground color to white.
2. Render some Clouds (choose Filter | Render | Clouds). Now render Difference Clouds (choose Filter|Render|Difference Clouds) twice. You should have an

image that looks about like Figure 2.33.

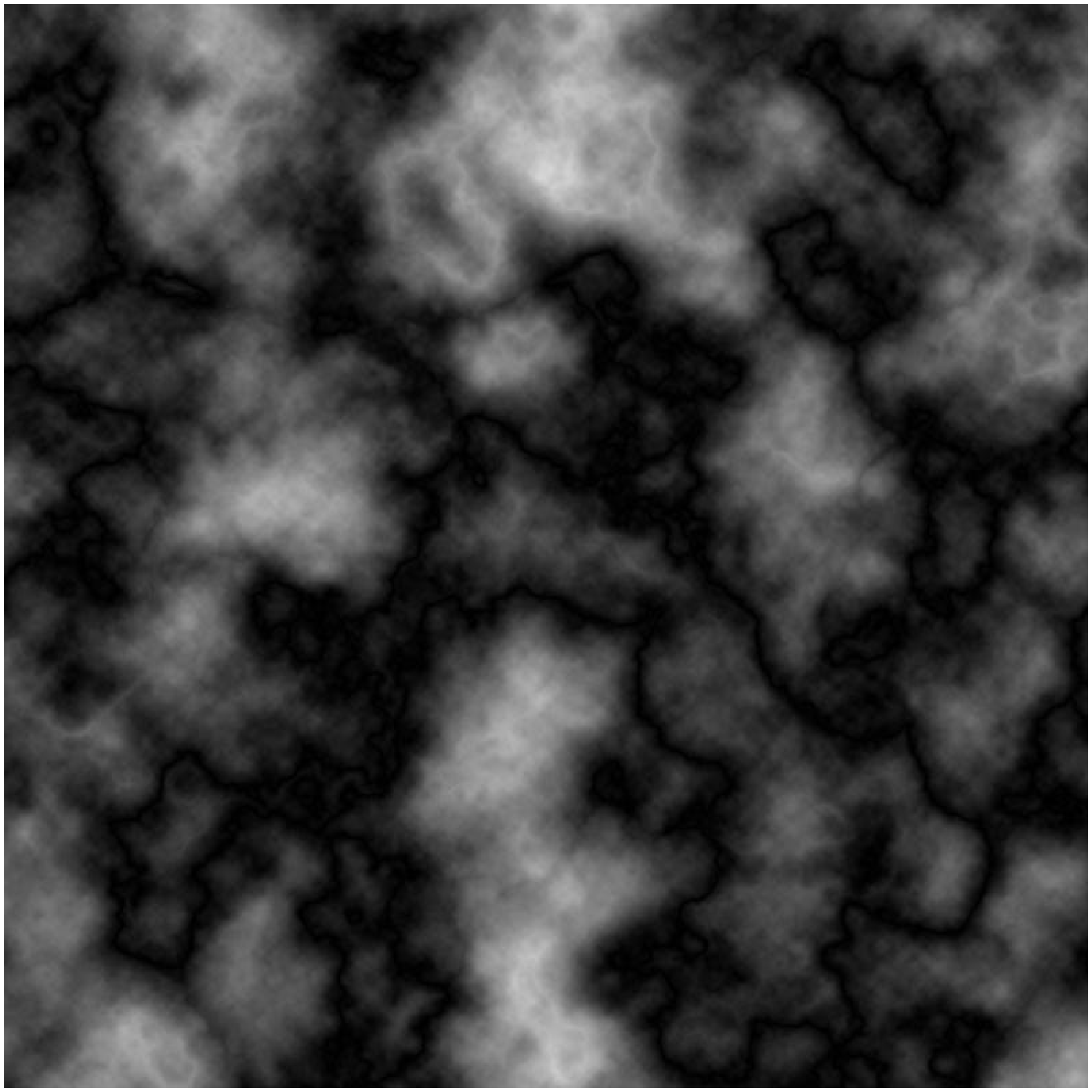


Figure 2.33 The start of metal, created by rendering clouds and difference clouds a few times.

3. Bump the contrast down a little, and then choose Image | Adjust | Brightness/Contrast. Set the contrast to -10.
4. Add noise by choosing Filter|Noise|Add Noise. Use the following settings: Gaussian, 20; Monochromatic. 5. Now add some blur. Choose Filter | Blur | Motion Blur. Settings: Angle 0 and Distance 15.
6. Then choose Filter | Sharpen | Unsharp Mask. Settings: Amount 50, Radius 5, Threshold 0.

Like most base textures, this does not look like much now (see Figure 2.34), but look at a real piece of brushed metal under even lighting and this is what you see. After we apply some effects to these bases in the next chapter and begin to build up a texture, you will see what can be done with a few simple base textures and some creativity.

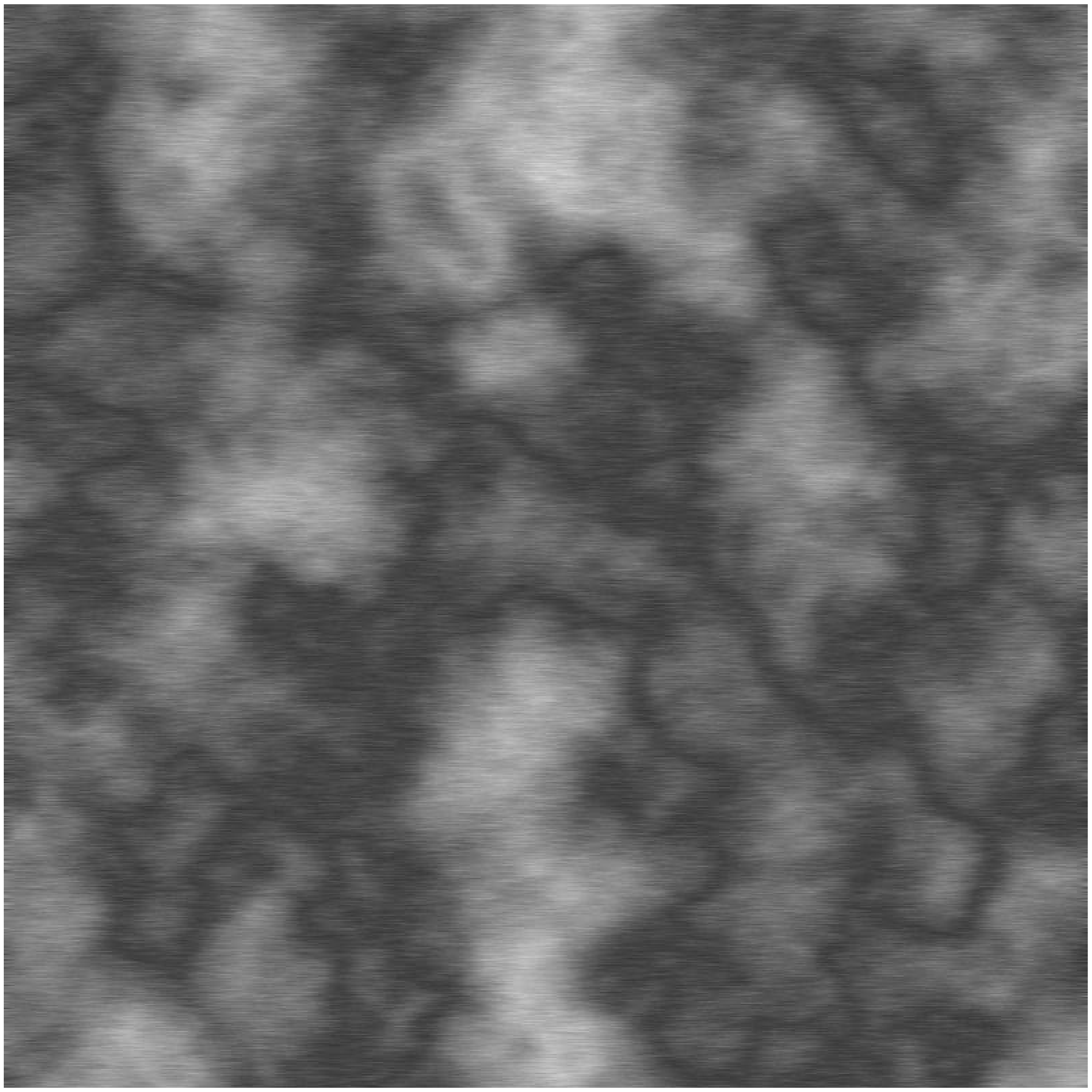


Figure 2.34 Our base brushed metal--not much now but we will build on it later.

Creating a Star Field

Our next base is a very useful one, a star field. With a few easy steps you can create some really nice stars for the sky. Start with an 800 x 800 image.

1. Create a new layer and fill it with black. Now add some noise, which will be the

stars. Choose Noise | Add Noise. Settings: Amount 40, Gaussian, and Monochromatic.

2. Now choose Image | Adjust | Brightness/Contrast. Settings: Brightness -55, Contrast +40.
3. Again, choose Image | Adjust | Brightness/Contrast. Settings: Contrast +50. You should see the stars pop out as in Figure 2.35.

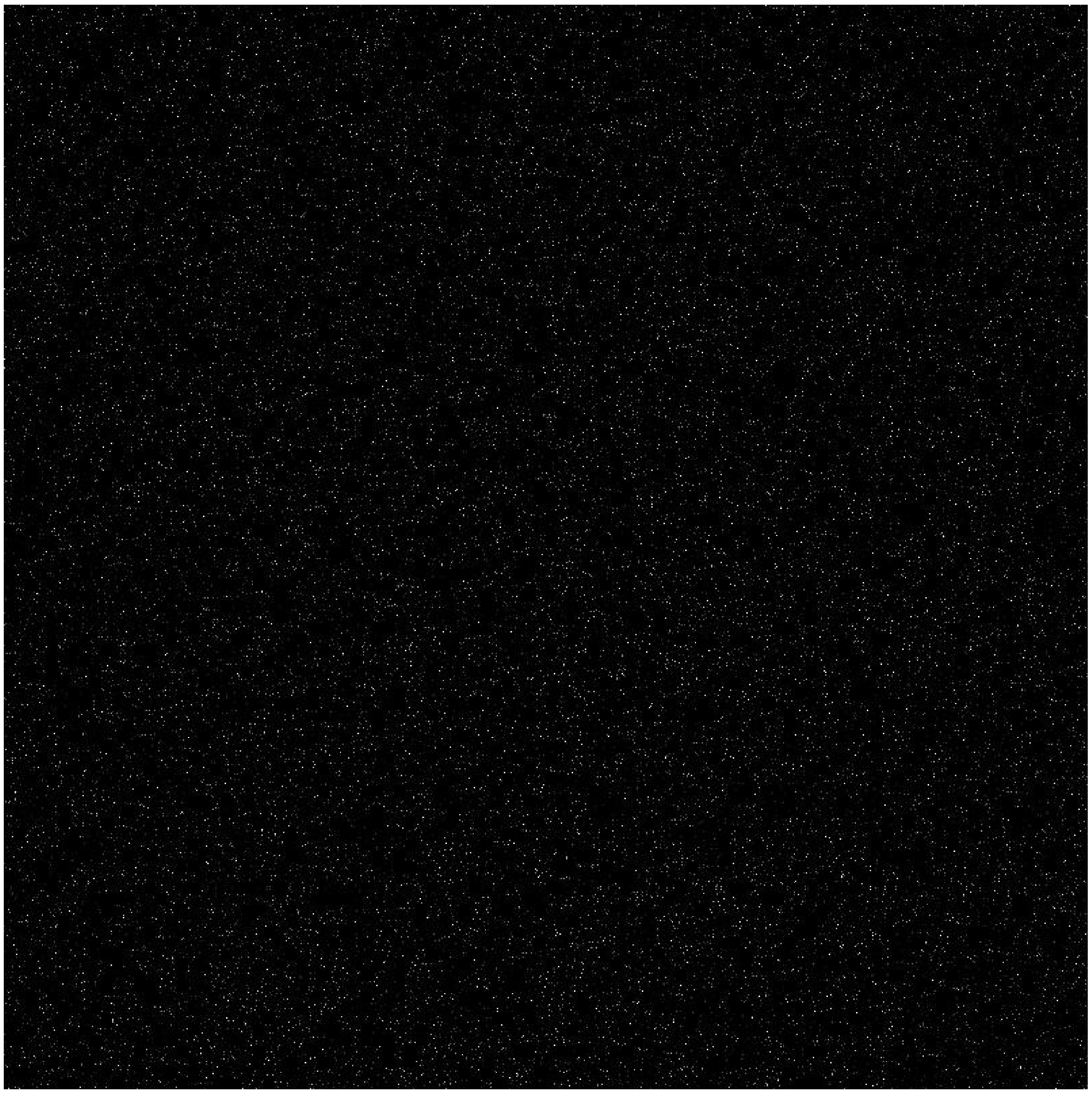


Figure 2.35 The stars at night are big and bright in your computer.

4. You can repeat Step 3 if you would like fewer stars, bigger stars, and brighter stars.

Creating a More Complex Star Field

Now we'll build on the basic star field and make it a little more complex:

1. Copy the star layer from above and increase the contrast dramatically. Choose Image | Adjust | Brightness/Contrast. Settings: Contrast +50.
2. Enlarge the layer by choosing Menu | Edit | Free Transform (Control "T"). Hold Shift to make the transform uniform, and drag out the layer a good bit beyond the edges of the canvas. Using this method you can make larger stars that are further apart and brighter. This adds more depth and realism to your sky.
3. Use the Magic Wand tool to select the black from this layer and delete it.

You now have a star field with more depth and variety (see Figure 2.36). Remember that you can move the larger upper layer about for randomness to the star pattern, and you can flip the layer vertically or horizontally.

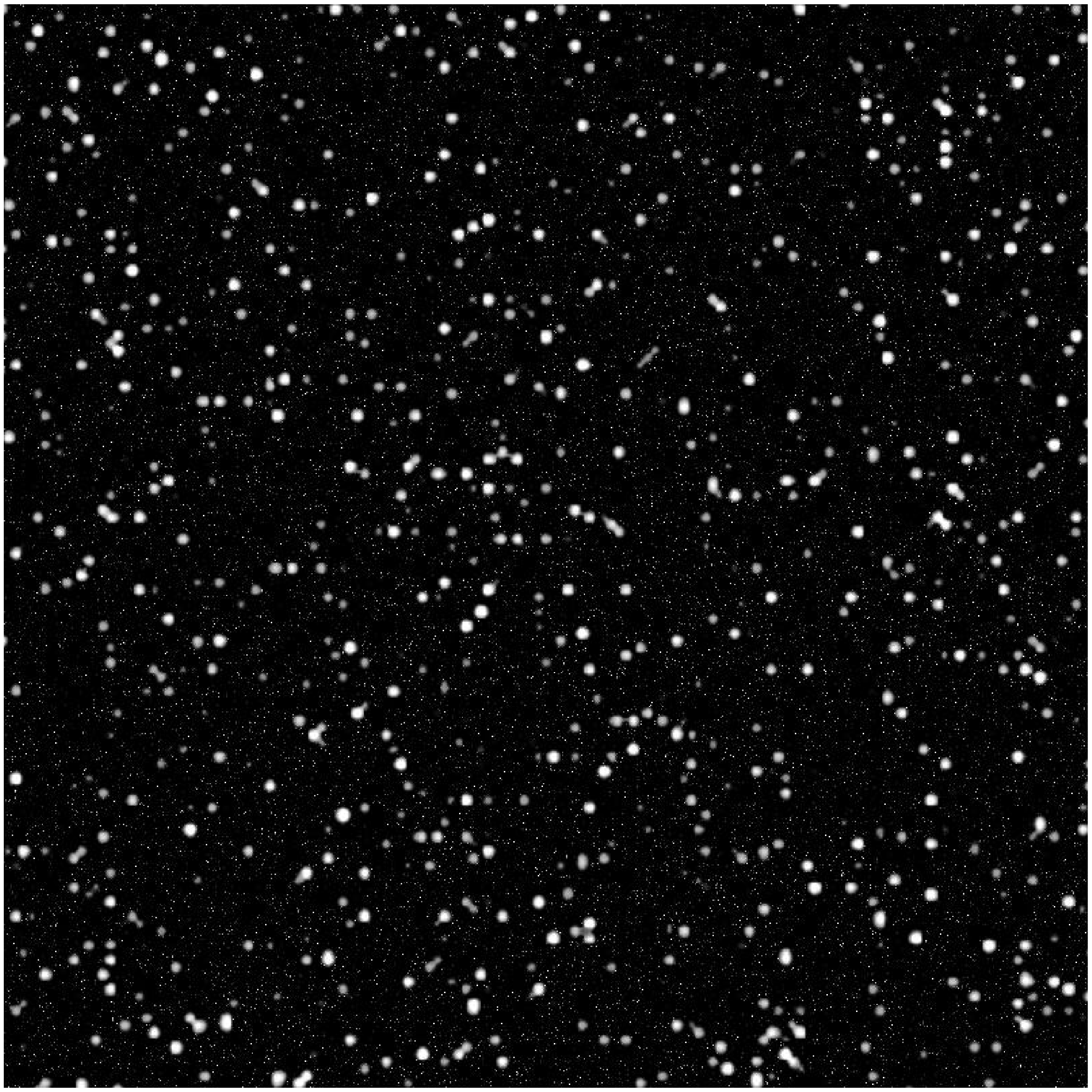


Figure 2.36 The deluxe star field; notice the larger, more random stars.

Creating a Deep-Space Star Nebula

Often the space scene in a game must be broken with some detail or feature, and a cloud nebula is a good one to use. You can also create a cold deep space look where our sun is but a distant light in a cloud-like galaxy with this effect.

1. Using your deluxe star field created above, create a new layer and set its mode to Screen.
2. Make your foreground color a bright blue and the background color black.
3. Use the Round Selection tool and set the feather to 25. Select an oval shape in the middle of the image.

4. Now render some clouds; choose Filter | Render | Clouds, and then Filter | Render | Difference Clouds. If you run the Difference Clouds filter a few times, you will get several variations that begin to look very nebula-like.

6. Now choose Filter | Distort | Twirl. Settings: 700. Your image should look like Figure 2.37.

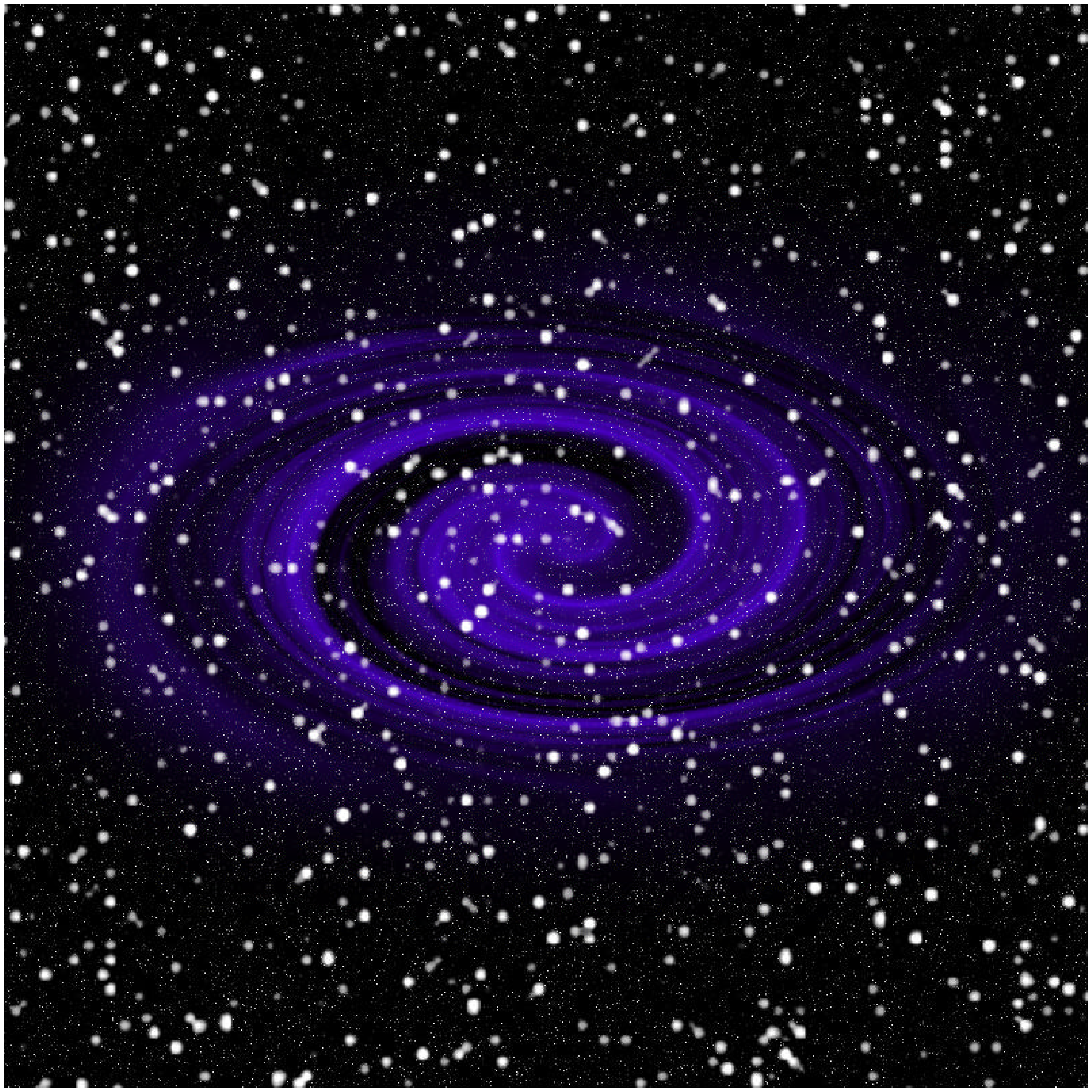


Figure 2.37 The beginnings of a universe.

7. Duplicate the nebula-cloud layer.

8. Do a Free Transform (Control “T”) and turn the layer about half a turn. Choose Image | Adjust | Brightness/Contrast. Settings: Contrast -40.

9. The final step is to put the distant sun in the center of your swirling galaxy. Create

a new layer and fill it with black. Set this layer on Mode | Screen.

10. To make the sun, choose Filter | Render | Lens Flare. Settings: 82%, 105 mm prime. By having the sun on its own layer, you can move it about easily and not have the lens flare rendered right over your galaxy image.

11. Now you can zoom into the rendered sun and use the Smudge tool to pull out a few solar flares (see Figure 2.38).

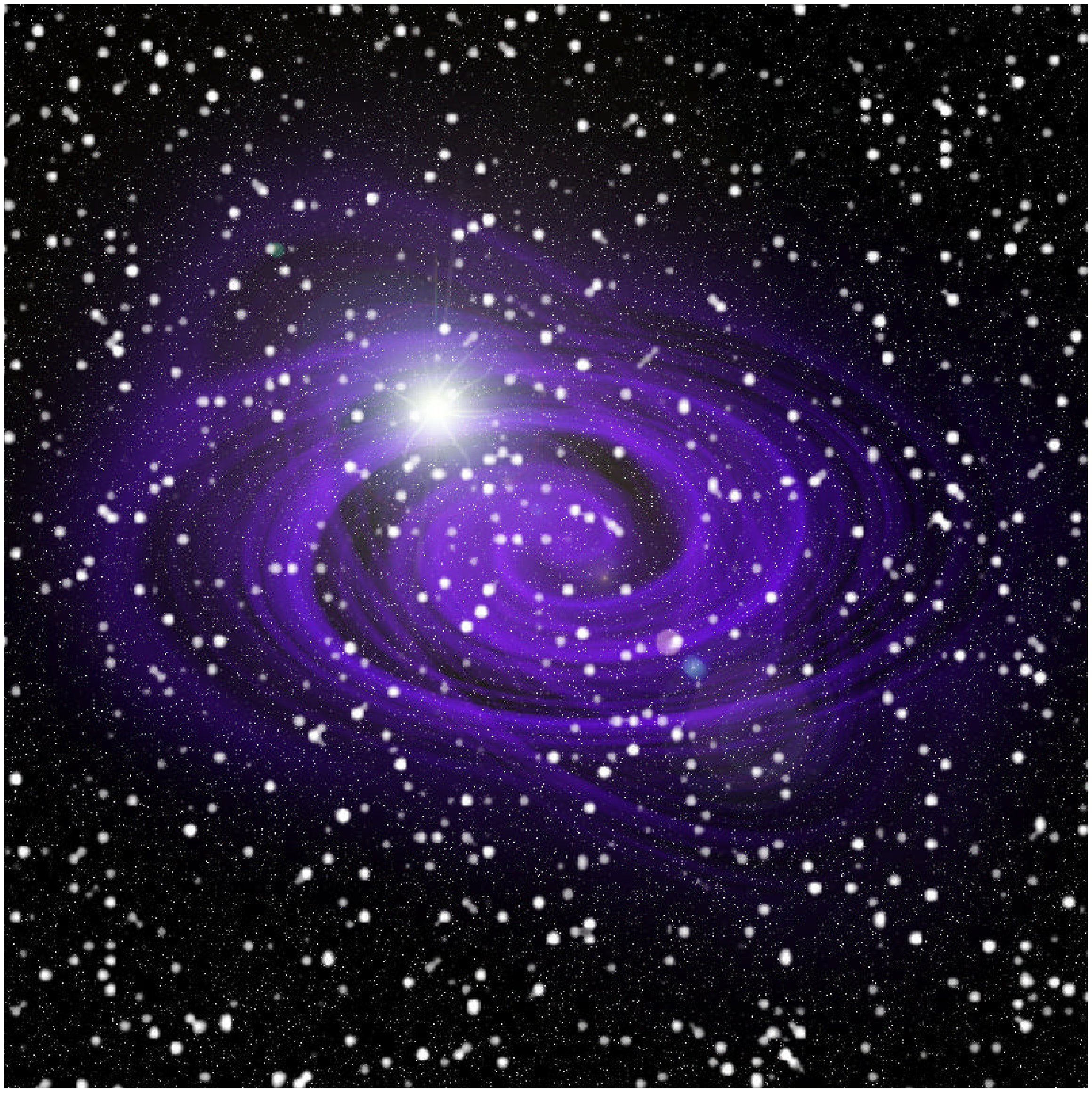


Figure 2.38 Deep space, with our sun small and far away.

Creating Wood

To create a passable wood texture, you can do the following in Photoshop:

1. Create a new image that is 256 x 256 pixels. Choose a light brown as the foreground color and a dark brown for the background color.
2. Run the Clouds filter (choose Filter | Render | Clouds), and then run the Difference Clouds filter several times until you have a good balance of pattern.
3. Optionally you can add some noise here (choose Filter | Noise | Add Noise). Make sure that you use a low amount and choose the Uniform and Monochromatic options.
4. Now add curves (choose Filter|Distort|Shear).

With some tiling you can make this a nice piece of wood. It is wood grained, but not very realistic (see Figure 2.39). This is where I depart Photoshop and go grab a digital image of some wood. I have included several images on the CD-ROM, and depending on what surface you are covering, you can choose from the old wood, weathered wood, pine, or one of the other files. Let’s use some of the tiling tricks we learned to make a tiling wood image that we can use later as a base for almost anything made of wood, such as tables, beams, telephone poles, and floors.

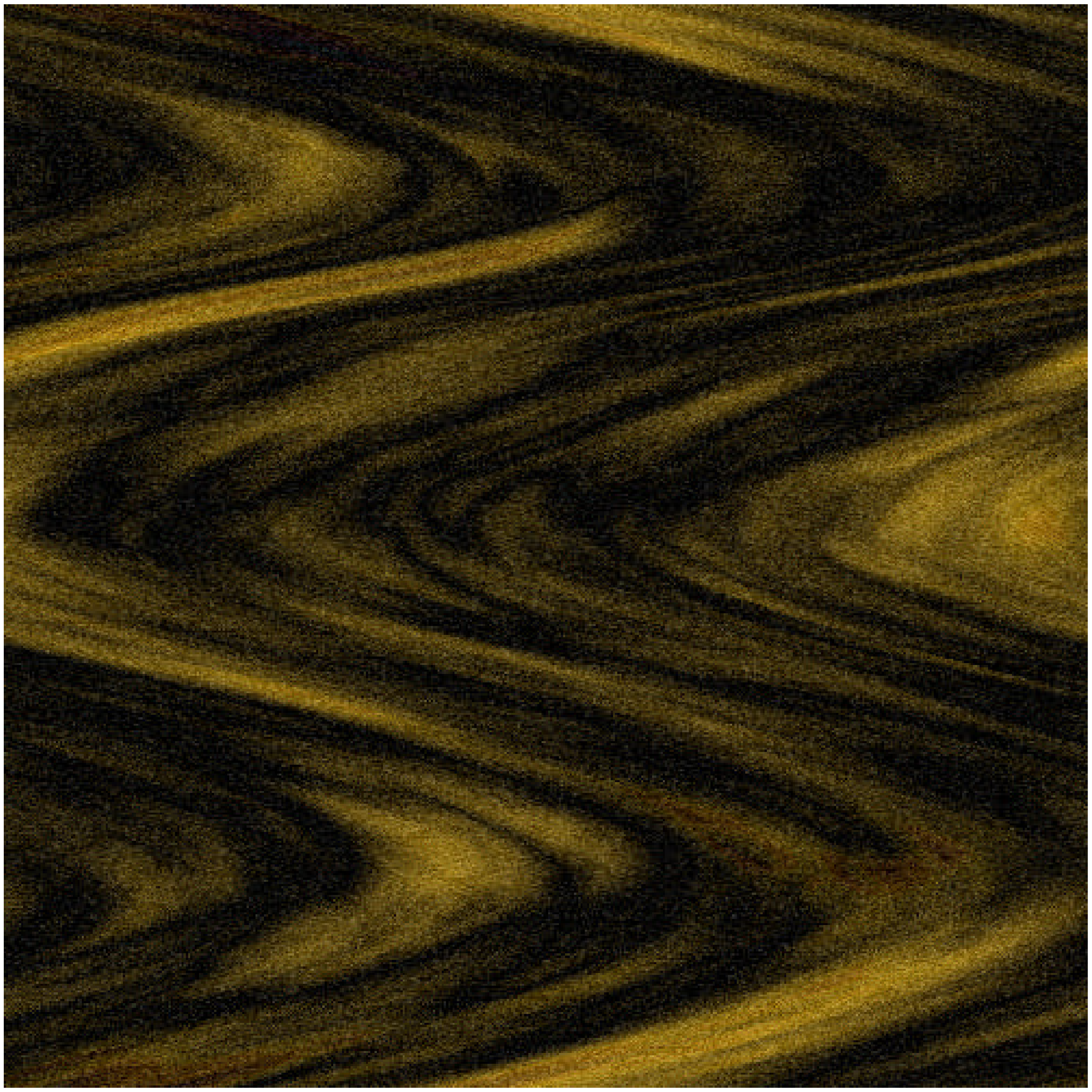


Figure 2.39 A passable wood grain, but not very realistic.

To create a base tileable wood for general purposes, start with the image named cedfence.jpg and do the following:

1. Make a duplicate of the image before altering it. Always keep your original images intact for future use.

   You may notice that this image is not perfectly square. Being *perfectly* square is almost always demanded by today's game technology. The reasons are basically mathematical; the computer can process a square image faster. This is referred to as the order of two, generally 32x32, 64x64, 125x128, 256x256, etc. Let's first make the image square.

2. Fit the image in your window by selecting the Magnifying Glass tool and right-clicking on the image. Select Fit In Window from the context menu. Now right-click again and select Zoom Out. This rather large image should now fit on your monitor with some room to work (figure 2.40).

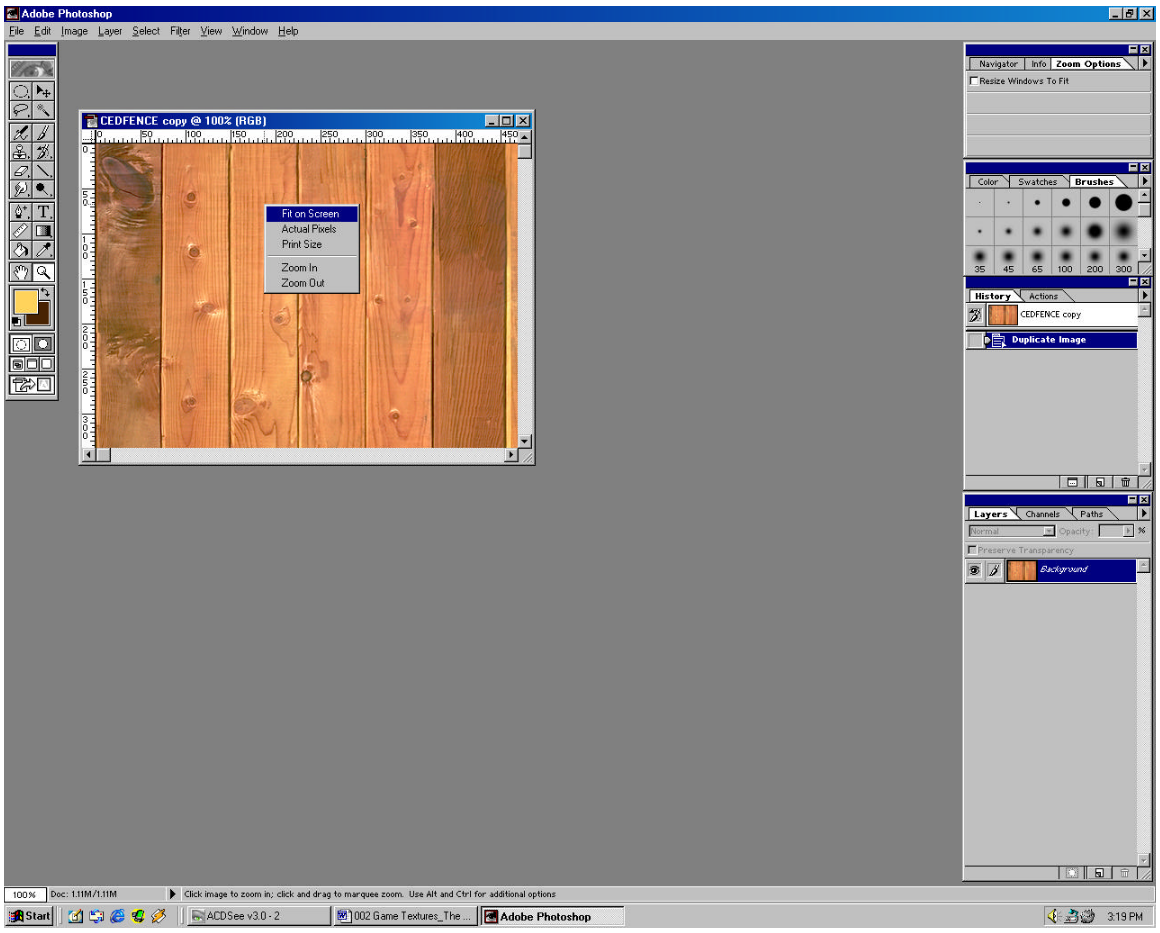


Figure 2.40 Even large images can be quickly sized to fit on your monitor so you can easily work on them.

3. Select the Crop tool from the toolbar.
4. Hold down Shift and drag out a crop area that covers as much of the image as possible without going off the canvas. Position the crop area on at least one seam of the boards and in an area you think will be easy to tile (figure 2.41). I am sure you already see that the seams of the boards don’t line up, that some boards are much darker than others, and that you generally have some work ahead of you.

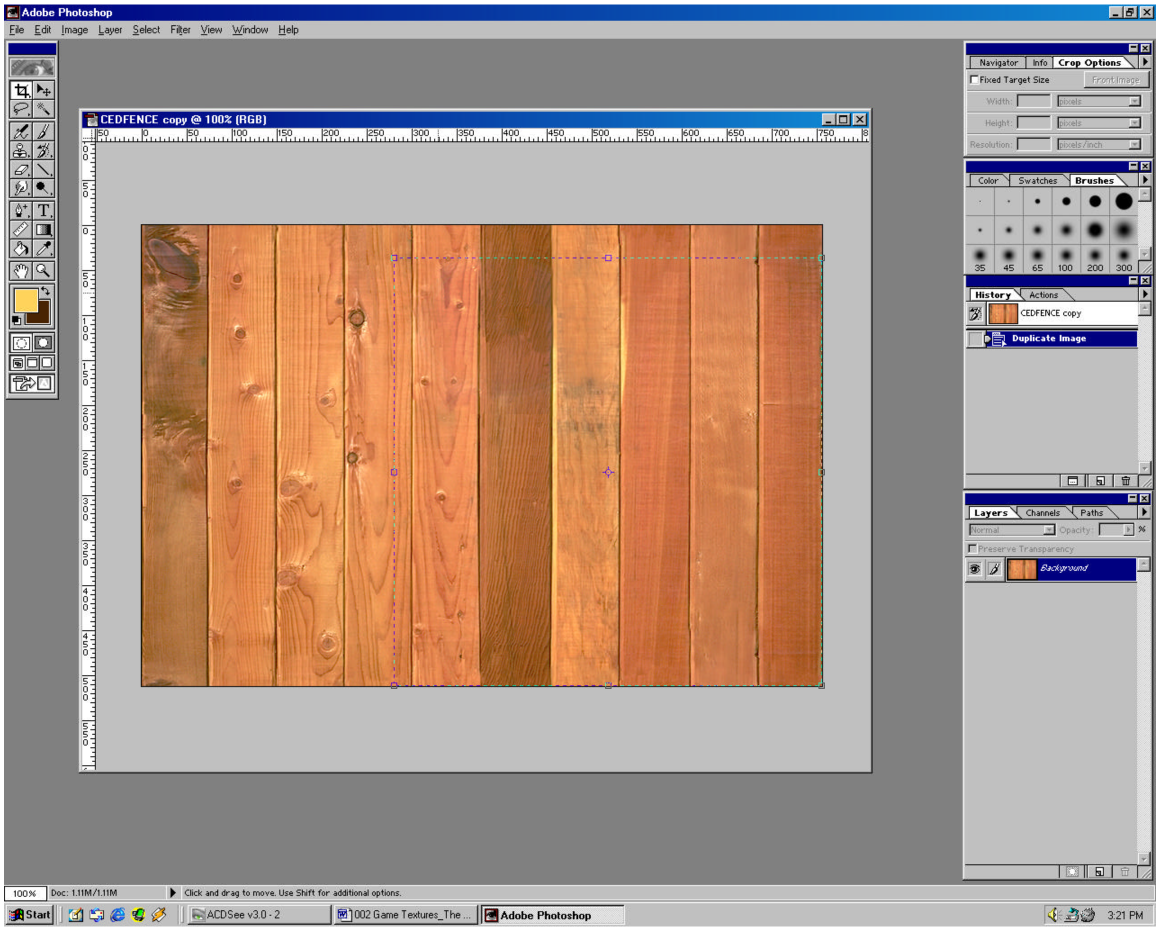
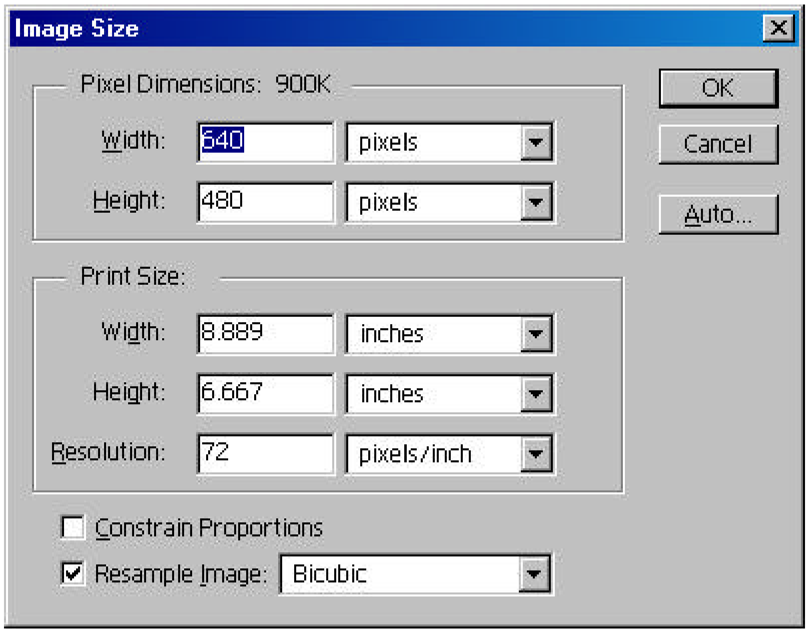


Figure 2.41 Cropping an image to find seams to help you with your tiling job.

5. Now that you have cropped the image, right-click on the header bar of the image and select Image Size. It does not really matter what size the image is, but it should have exactly the same pixel height and width. If the image size is off by one or two pixels, go ahead and Un-select the Constrain Proportions box (figure 2.42) and type in the right number. For example, if the image is 512x513, change the 512 to 513. This one-pixel stretch of the image won't do any damage to the image, but if your measurements are off by more than a pixel or two, go back to Step 5 and make sure you are holding down Shift when you define the crop area.



Pixel Dimensions: 900K
Width: 640 pixels
Height: 480 pixels

Print Size:
Width: 8.889 inches
Height: 6.667 inches
Resolution: 72 pixels/inch

☐ Constrain Proportions
☑ Resample Image: Bicubic

Figure 2.42 The Constrain Proportions box unchecked so you can size an image to exact pixel sizes.

6. Duplicate the layer and make sure the new layer is selected.
7. We can use this particular texture as floorboards, and although the texture will have to tile in two directions, the seam of the tile can be matched to the seam in the image and eliminate a lot of work for us. In other words, we effectively only have to worry about tiling in one direction because we have seams to help us on the other direction. But we still have to look at the seams and make sure they meet cleanly. To make the seams of the board meet cleanly without changing the size of the image, we will do a Free Transform (holding down Shift so it is uniform) and drag out the layer a bit larger than the image. Do this until the seams meet up at the edges perfectly, as in figures 2.43 and 2.44.

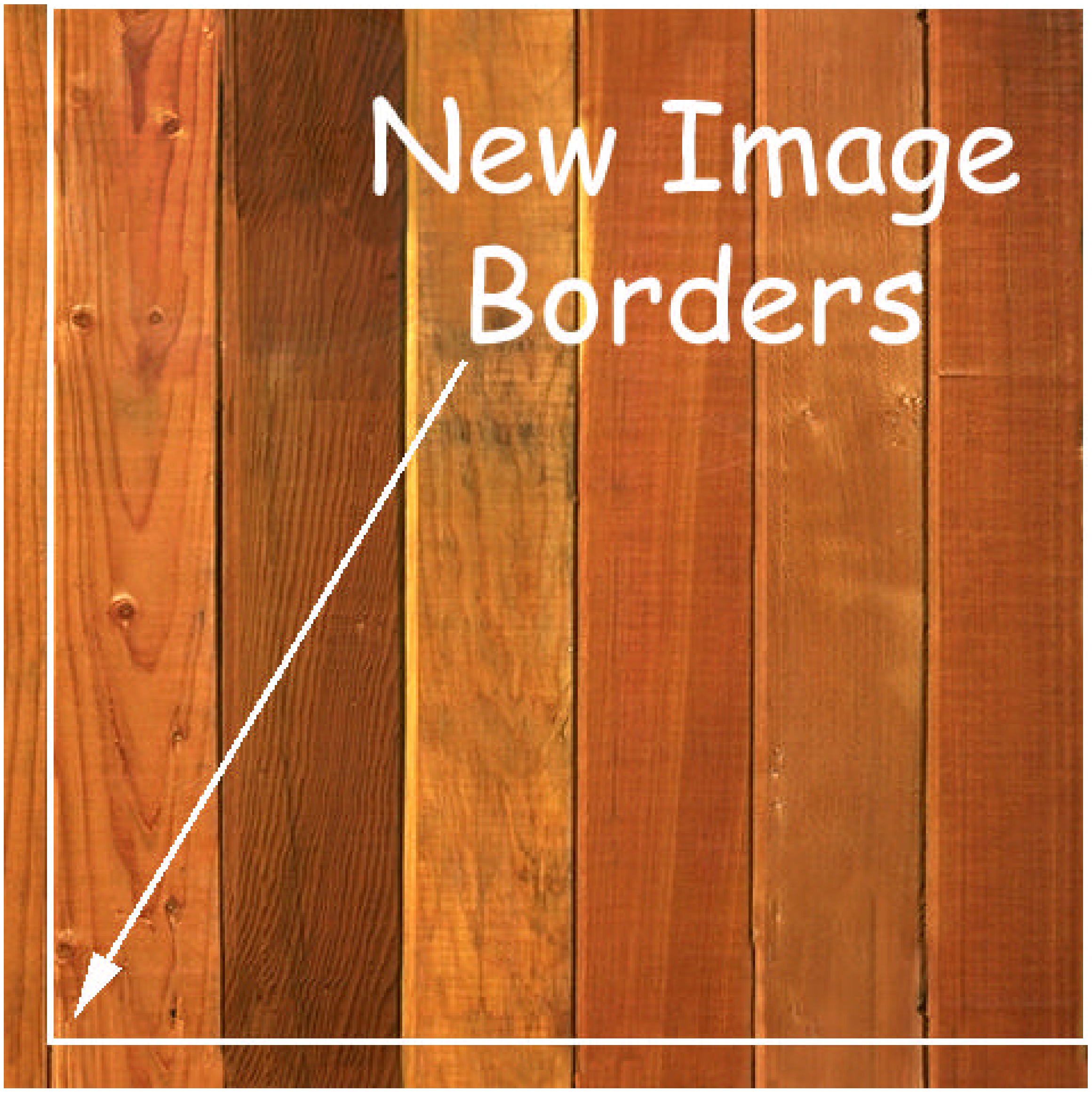


Figure 2.43 The seams of the board do not naturally meet the edge of the image so we must use Free Transform to match the seams we want to use with the edge of the image.

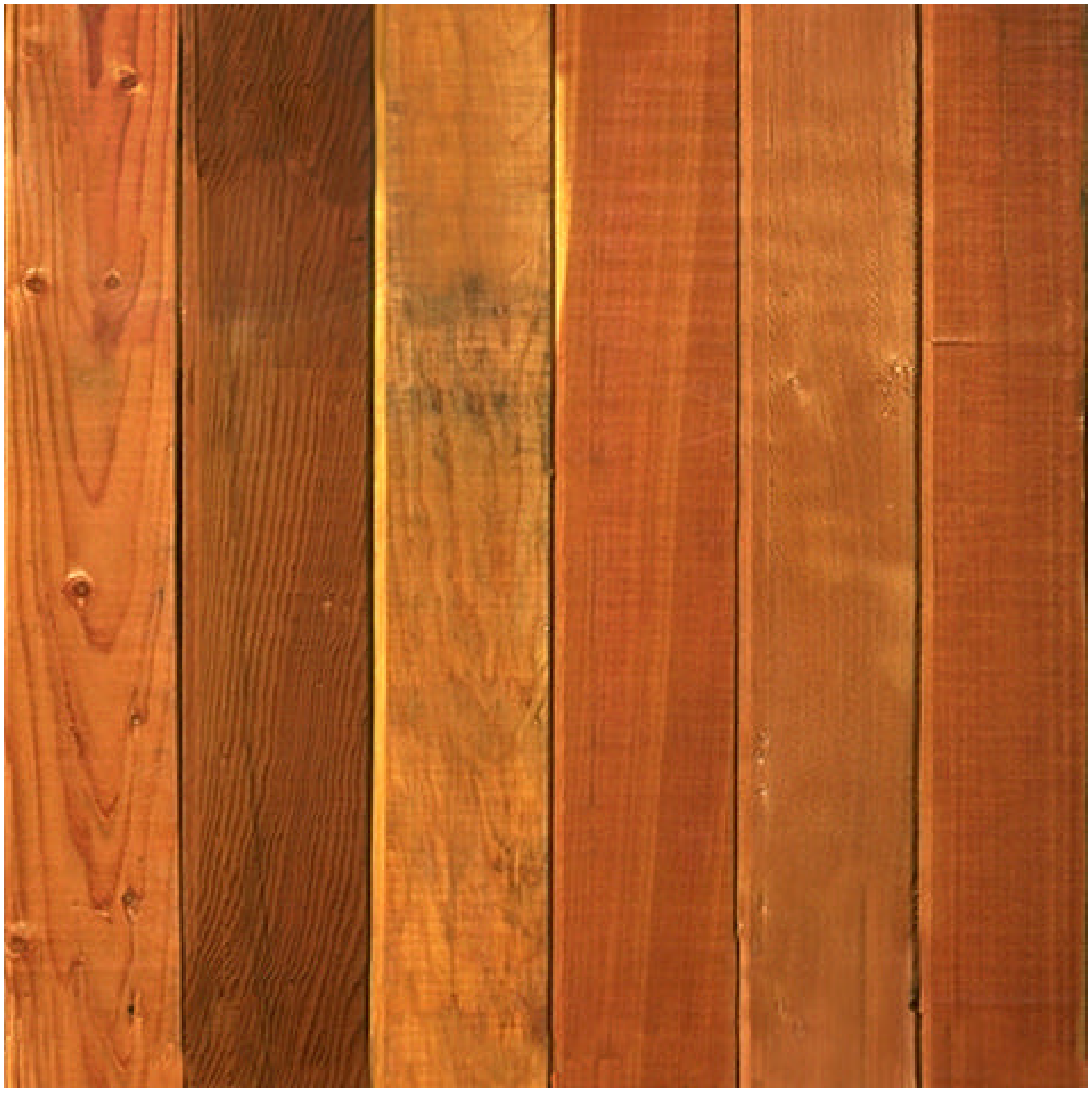


Figure 2.44 The image Free Transformed to make the seams of the boards meet the edge of the image. The image size is the same, but the portion of the image we want to use was made larger to fill the image.

8. Crop the image again and check that it is perfectly square.
9. Go back to “Tips for Horizontal and Vertical Tiling” earlier in this chapter, and tile this image. It is suggested that you copy the top of the image and flip it and drag it down. Erase the seam and then flatten the image and repeat the process for the sides. The last step is to use the Clone tool and a larger, very soft brush and clone out the knots in the wood. Test the image by using the Define Pattern and Fill | Pattern method as described above.
10. A final note: If you are making this base texture for a certain type of game or

setting, keep that in mind and adjust the color balance, contrast, and other settings to reflect your world.

Creating a Stone Texture

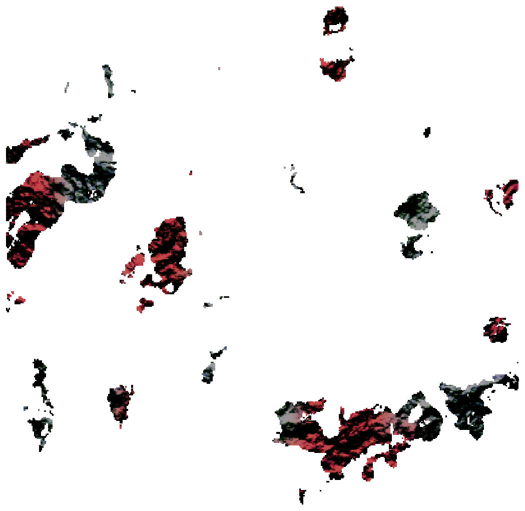
In this exercise, we'll use Photoshop to create a stone texture:

1. Start with a new image that's 800x800.
2. Select a dark gray for your foreground and a light blue for the background.
3. Choose Filter | Render | Clouds.
4. Choose Filter | Noise | Add Noise. Set the amount to 3, Gaussian, Monochromatic.
5. In the Channels window, create a new channel and select it.
6. Choose Filter | Render | Difference Clouds.
7. Choose Filter | Noise | Add Noise and keep the same settings.
8. Choose Filter | Render | Difference Clouds, and press Ctrl + F to repeat the action until the channel is balanced with black and white as shown in figure 2.45.

Figure 2.45 The image ready to make stone as it is a balance of patterns.

9. In the Layers window, select the layer we were working on before. Choose Filter | Render | Lighting Effects. Settings: Intensity 59, Focus 69, Properties: Gloss 100, (Shiny) Material 48 (Metallic) Exposure 6, Ambiance 4. Set the Texture Channel, and the height to 100.
10. Adjust the direction of the light to your liking.
11. Adjust the contrast and brightness (choose Image | Adjust | Brightness/Contrast). It will most likely look best with the Brightness and Contrast up a bit.

Experimenting with these sets of effects, you can make all sorts of stone and even surfaces like dinosaur or troll skin. See figures 2.46 through 2.48.



Figures 2.46 The finished stone texture with burned in red streak to look like dinosaur skin.

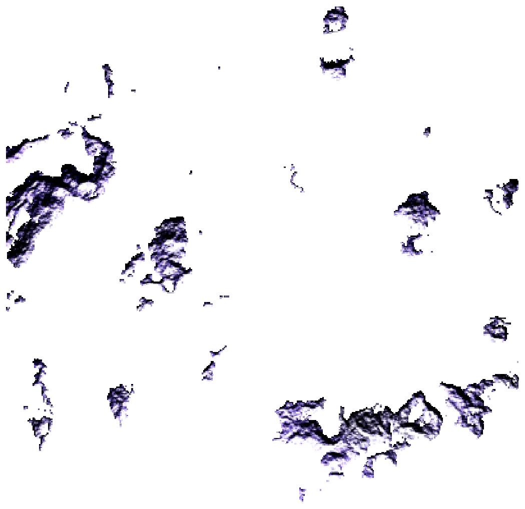


Figure 2.47 A wet cave wall with light from below.

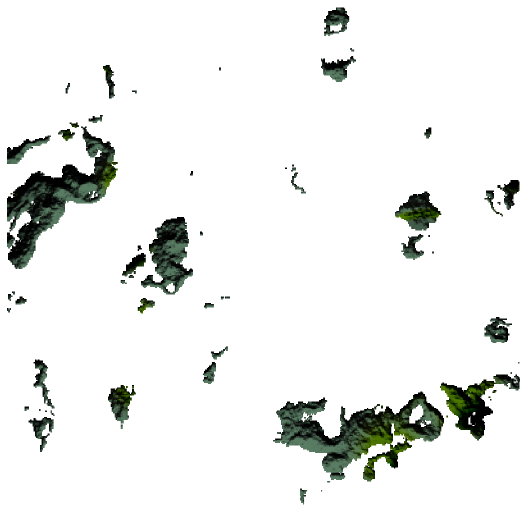


Figure 248 A mossy cave wall or more dinosaur skin? Depends how you use it.

Wetting the Stone

Now let’s go one step further and make the stone look wet:

1. To make the stone look wet, copy the texture layer before you do Lighting Effects.
2. Apply Lighting Effects to the bottom layer and keep it matte or non-glossy.
3. Then apply the same Lighting Effects to the top layer but turn up the glossiness. Now, while you are on the top layer, choose Layer | Add Layer Mask | Reveal All.
4. You should use the Difference Clouds filter on the layer mask a few times and then adjust the Contrast and Brightness.

Creating Quick Mud

To make a gooey wet mud texture quickly, do the following in Photoshop:

1. Create a new 600 x 600 image. Press the D key to set your foreground and background colors to black.
2. Apply the Clouds filter to the layer (choose Render | Clouds), and then apply the Difference Clouds filter to the layer. Keep applying this filter (Ctrl + F) until you have a very organic image, veined with black.
3. Apply the Emboss filter (choose Stylize | Emboss). I set the Height to 6 and the Amount to 220%.
4. This is the step that makes this look like wet mud. Create a duplicate of the layer and use Gaussian Blur on it, use 3 pixels. Invert the colors and set the layer mode to Overlay.
5. Adjust the hue and saturation (choose Image|Adjust | Hue/Saturation) and make sure the Colorize box is checked. Bring the hue and saturation down a bit. You

can also adjust the Brightness and Contrast of this layer and get different ground effects. I was able to achieve a nice dusty rock as you would find in a desert; see figure 2.49.

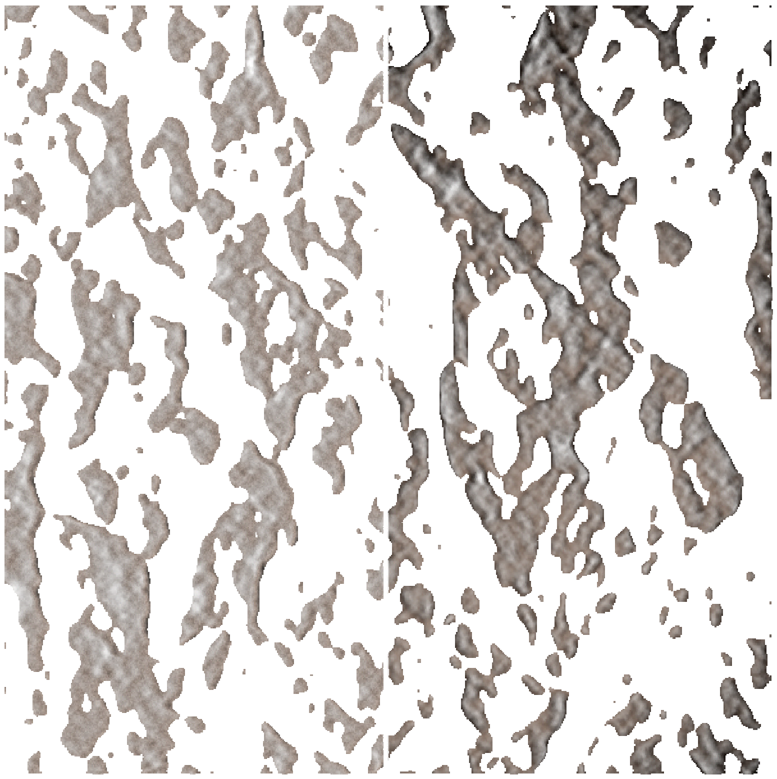


Figure 2.49 The dusty desert and wet mud textures.

Creating Parchment

In this exercise we will create a paper parchment texture, complete with ripped edges. We are creating this texture primarily to be used in a game. To create a parchment texture for high polygon/high-resolution 3D work, you would want to have a more subtle texture, maybe even scanning real parchment. For game purposes, you are usually creating images that will be color-reduced and shrunk dramatically, so you have to try to compensate for the loss of quality. Therefore, we will create a parchment texture that is rather rough to the eye, but will look great in a game.

To create parchment texture, follow these steps in Photoshop:

1. Create a new image, 600 x 600, and make the background white.
2. Create a new blank layer. Fill this new layer with a golden yellow (I used RGB 164,143,0).
3. Run the Texturizer filter (choose Texture | Texturizer) and select the Sandstone

texture. Set scaling at 100% and Relief at 10. Leave the light coming from the top.

4. Choose Filter Brushstrokes | Crosshatch. Set the Stroke Length to 9, the Sharpness to 2, and the Strength to 1. Your image should look like figure 2.50.

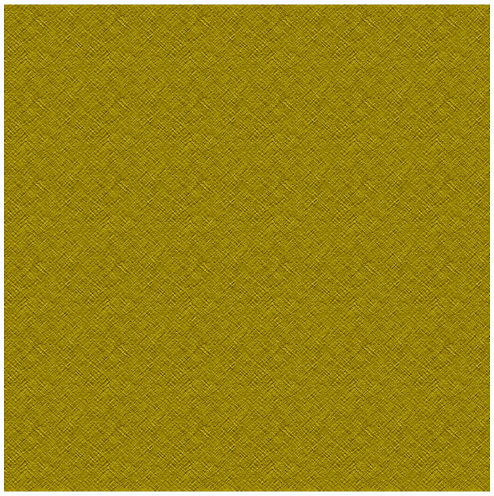


Figure 2.50 The paper texture shaping up.

5. Create a new layer. Here we will define the outline of the paper. Use a hard and narrow brush, and sketch the jagged outline of your page using pure black.
6. Fill the area outside the page area with black. You may notice a small outline or halo effect of the yellowish color around the fill area; all you have to do is hit the paint bucket a time or two and it will fill over this defect. You will now have an image that resembles the final parchment page in outline, as shown in figure 2.51.

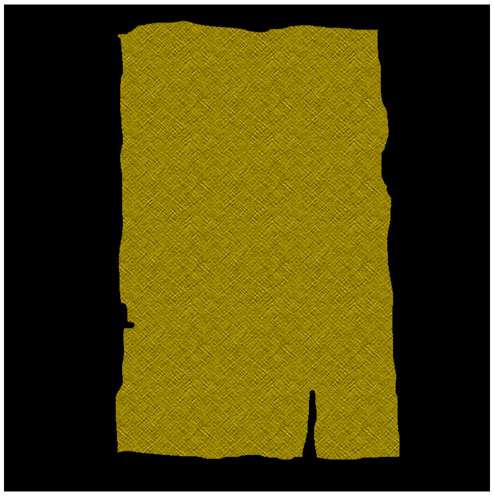


Figure 2.51 The black area around the parchment page is actually a layer over it defining the edge of the page.

7. Use the Magic Wand tool to select the black area, go to the paper layer, and cut away the outer edge of the paper. You can now hide or even delete the black outline layer.
8. To make the paper look rough on the edges and to finish the texture of the paper itself, run the Spatter filter (choose Brush Strokes | Spatter). Set the spray radius to 10 and the Smoothness to 5. See figure 2.52 for the completed texture.

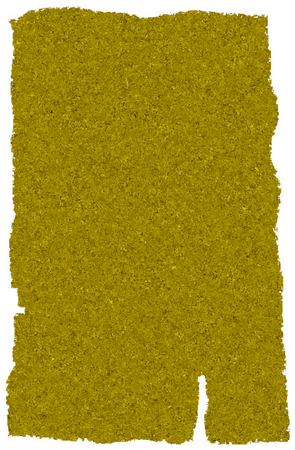


Figure 2.52 The completed page of parchment.

If you need to use this in a game as a sheet of paper, we will look at that later because it involves different steps than it would if you wanted to use this paper as a basis for a Web site, where you can go in and age the paper, write on it, and add a drop shadow. We'll look at these aspects of taking our textures further in the coming chapters.

Project: A Space Base Wall

Let's use the base textures we made to create an actual game texture. We will start with the standard space base wall.

1. First open the rust texture from the earlier exercise. As always, duplicate the original so you always have it to work from.
2. Flatten the image. Go to Layer and choose Flatten Image.
3. Duplicate the background layer and name it 'Trim'. Apply the Drop Shadow effect (leave the settings at their defaults), and apply the Bevel and Emboss filters. Set the Style to Emboss and the Depth to 3 and the Blur to 2.
4. Select a huge portion of the middle of the image, leaving the bottom area a little larger than the top. Delete this area from the image. Already we have a cool rusted panel wall (figure 2.53).

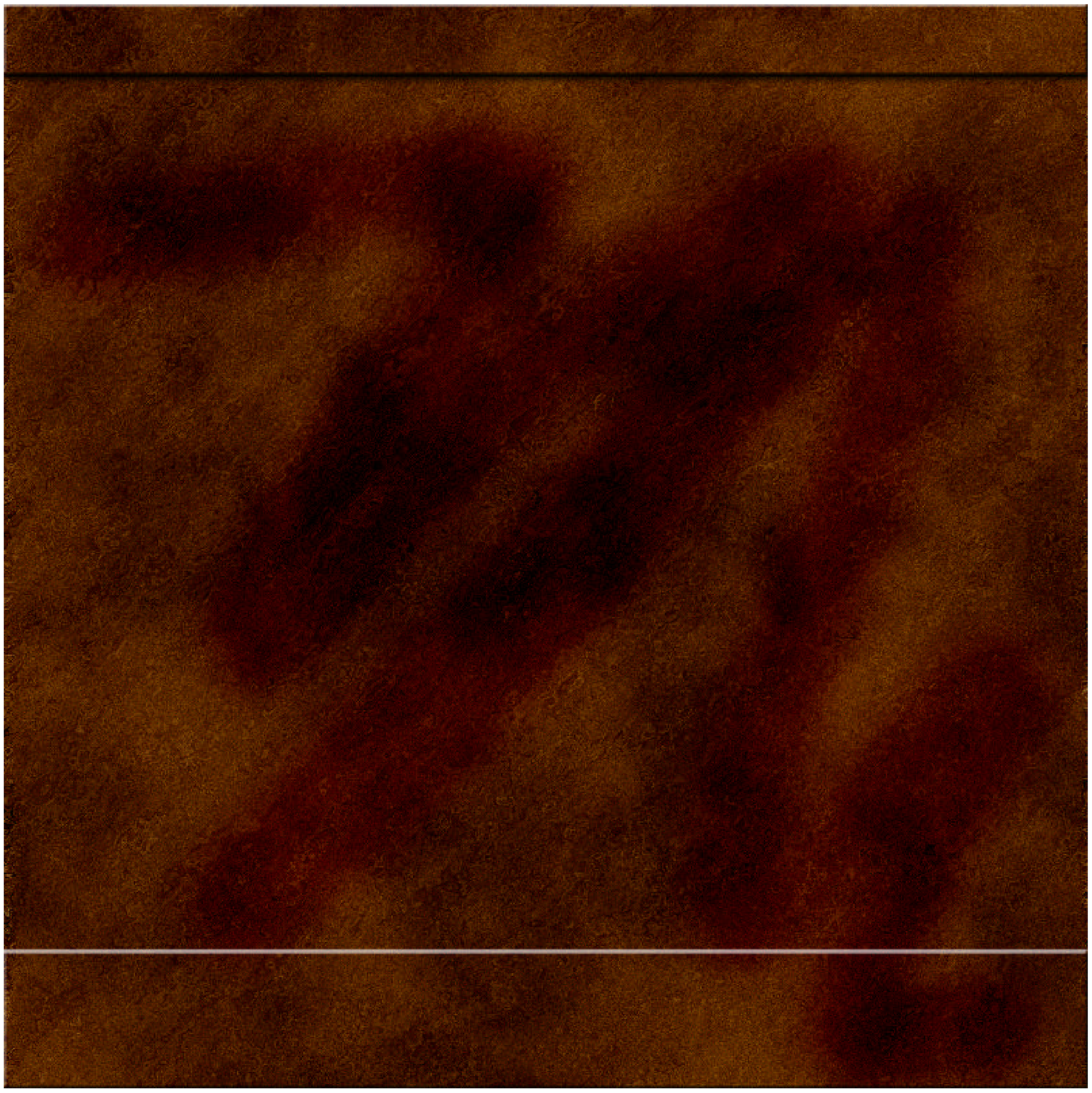


Figure 2.53 Already we have a cool rusted panel.

5. Turn on the grid. Since this is a 600 x 600 image, set the grid spacing to every 60 pixels. This will divide the image evenly and make your work easier. Since one of our goals is clean tiling, we should always work this way.
6. Now use the grid to select a square from the middle of the panel on the background layer. Paste the square into its own layer and name it ‘panel’. It should appear centered in the new layer. Copy your effects from Layer 2 and paste them here by right mouse clicking on the layer and choosing Copy Effects from the context menu. You paste effects in the same way by right clicking on the layer and choosing Paste Effects from the context menu. Your image should look like figure 2.54.

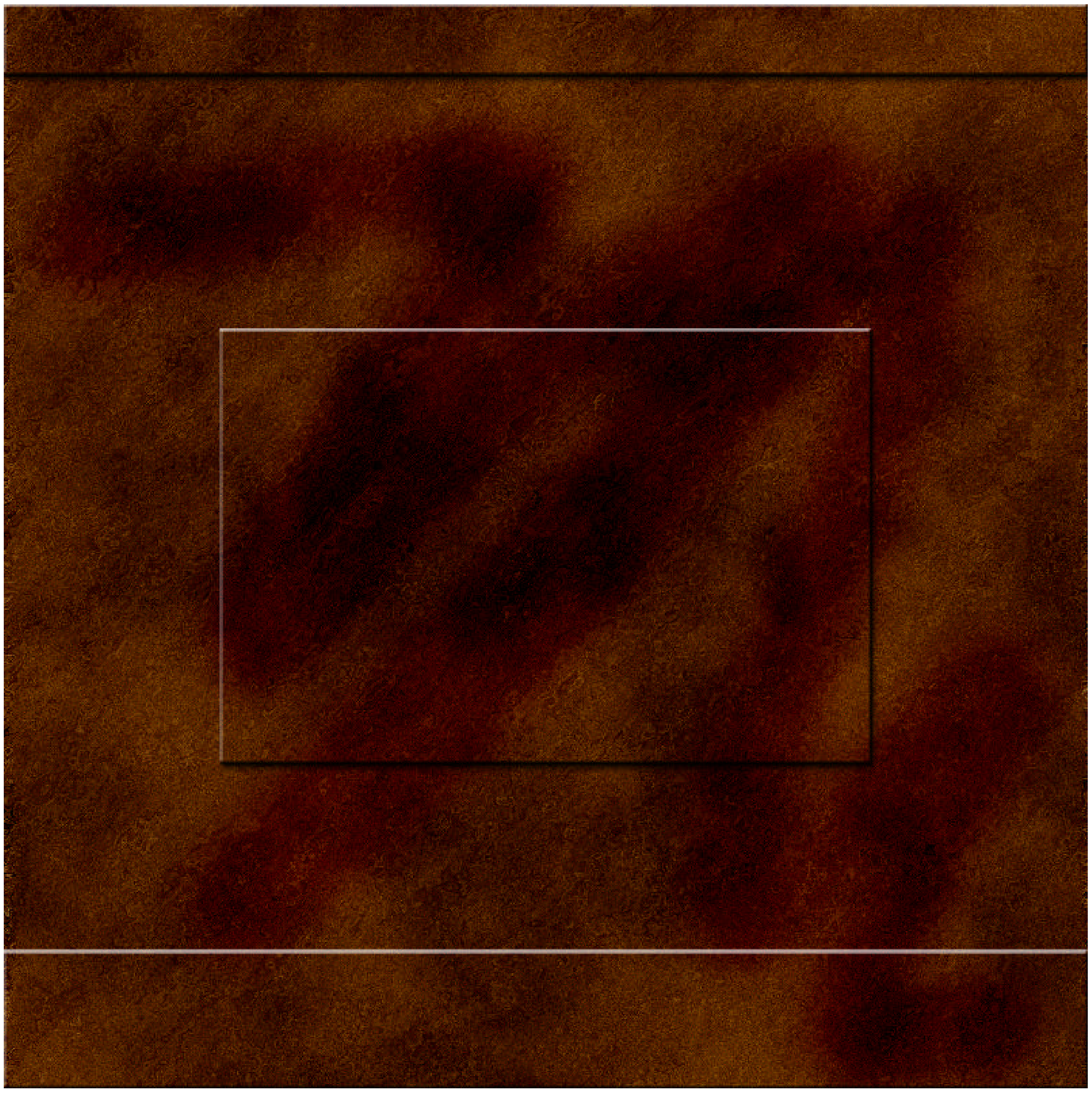


Figure 2.54 Notice how the grid evenly divides the image and how the new panel we just created sits neatly in the center.

7. You can add bolts to metal very easily. Create a new layer and name it ‘bolts’. Paste the same effects from the previous layers into it (for speed’s sake, I sometimes simply duplicate a layer that has the effects I want to use and simply Select All and Cut). You will want to zoom in a bit here as you work. Set your grid lines to appear every 20 pixels, and turn the grid on if you had turned it off.
8. Select a color similar to the wall color, maybe a bit lighter. Select a small hard brush and simply paint in a circle (which magically appears to be a bolt due to the effects) every few grid spaces, as in figure 2.55.

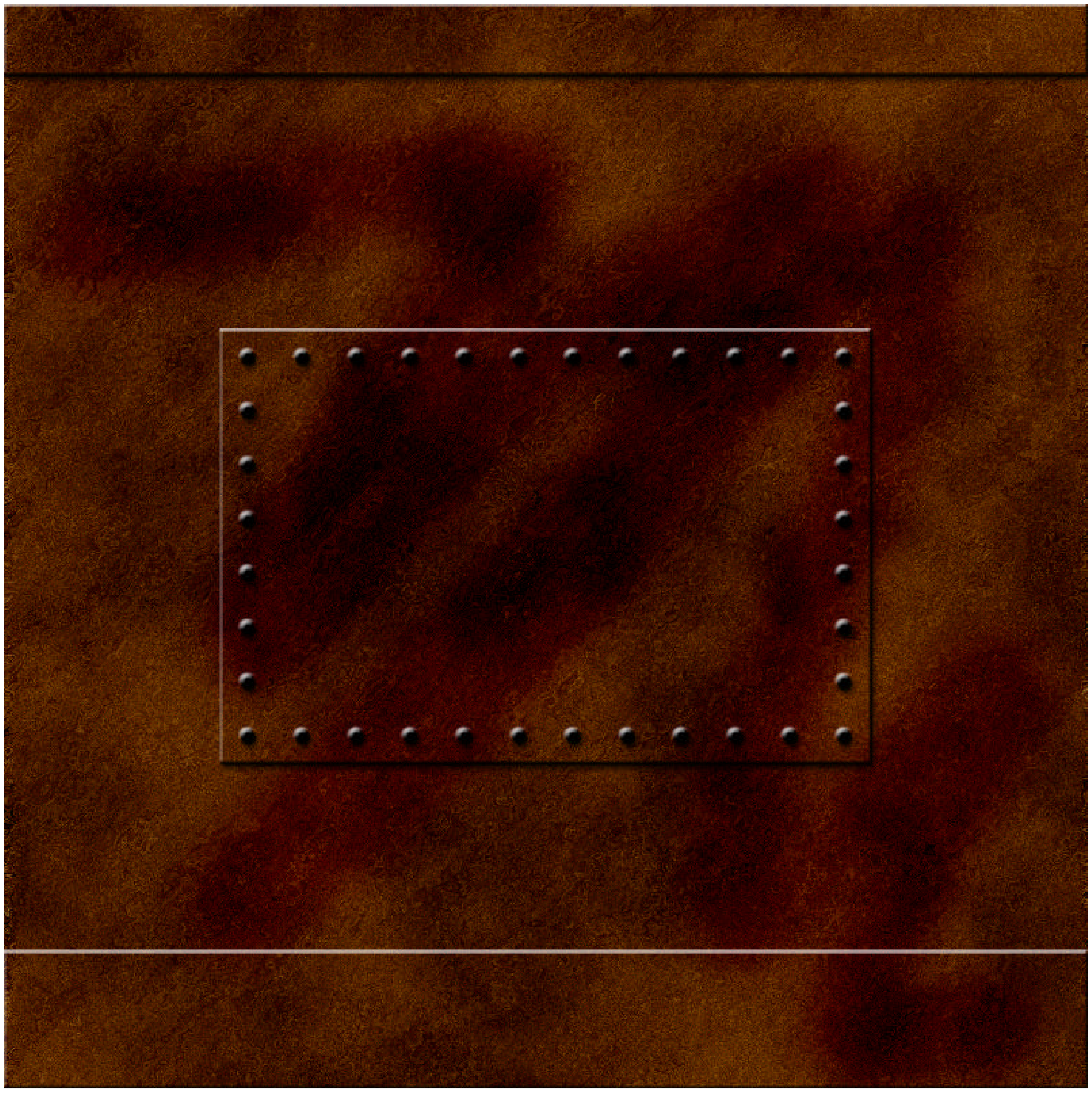


Figure 2.55 Bolts are made easy with layer effects in Photoshop.

9. Go back to the ‘panel’ layer, drag out two guides, and make sure they snap to the center of the panel. Make a roughly square, or horizontally rectangular, selection inside the panel. Use the guides to make sure the selection is snapped to center, and copy the selection. Paste the selection into a new layer and name it ‘vent cover’.

10. Create a new layer under the ‘vent cover’ layer, but over all the previous layers. Set this layer’s mode to Color Burn and take it down to about 23%. Airbrush some black lines coming from the vent and going in all directions. You can also apply a Radial Blur to this layer to help make the burns look better. You can see in figure 2.56, and in the image included with this book’s companion CD-ROM, that I also added some drips to the wall.

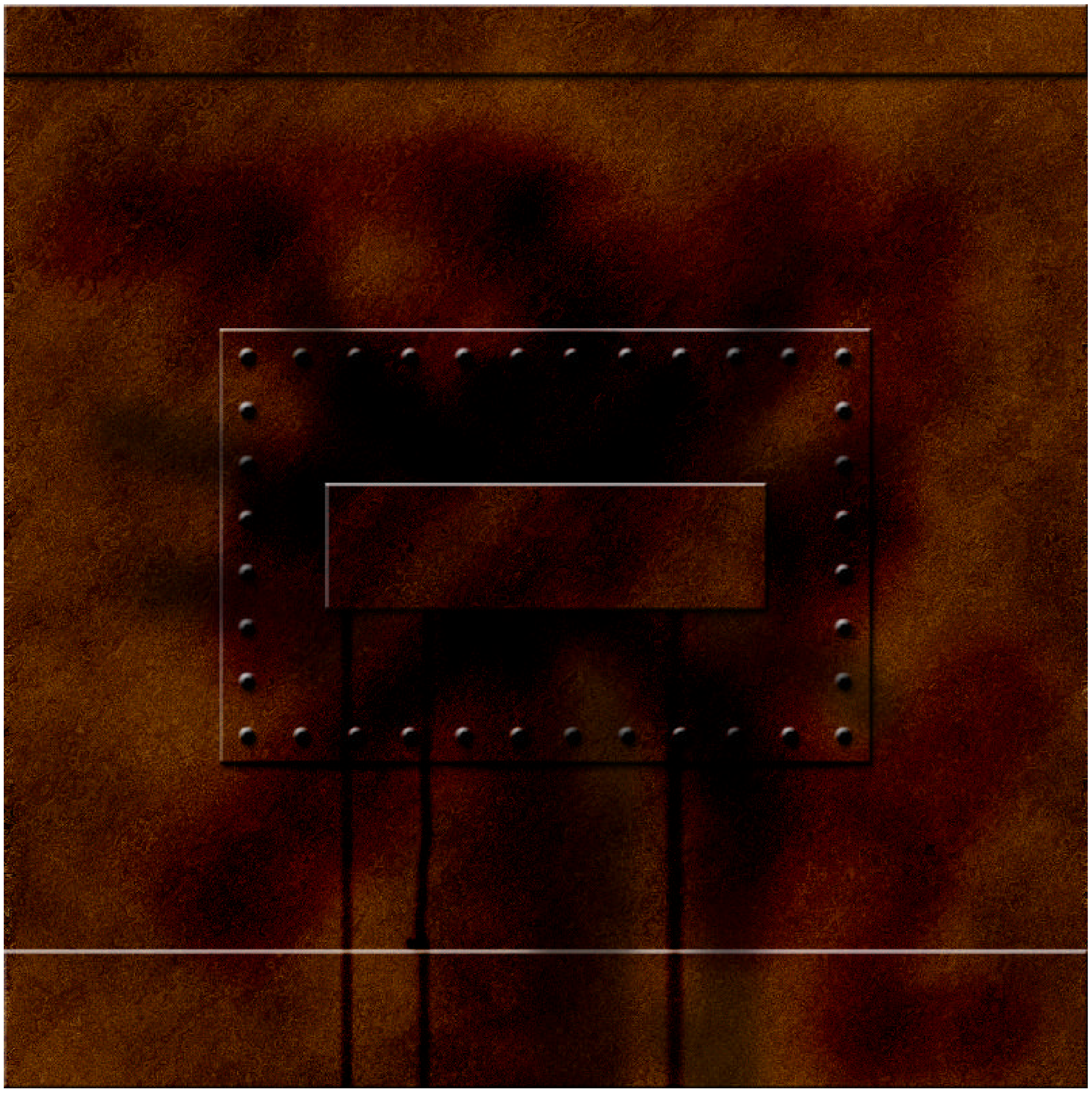


Figure 2.56 Notice how the blast marks coming from the vent and the drips add to the texture.

11. Create another layer and name it ‘decoration’. You can do anything you want here, but what I did was to select a few areas, fill them with red, lower the opacity to around 50%, and apply the Spatter filter (choose Brush Strokes | Spatter).
12. For a final, menacing touch, I added a skull. Simply set your foreground color to black, add a text layer, and then paste in the effects you copied earlier. The effects will still be in RAM so you don’t need to recopy them. Take the opacity down to about 50%, and type in the capital letter ‘N’ and use the Wingdings font. See the results of this 12-step texture exercise in figure 2.57.

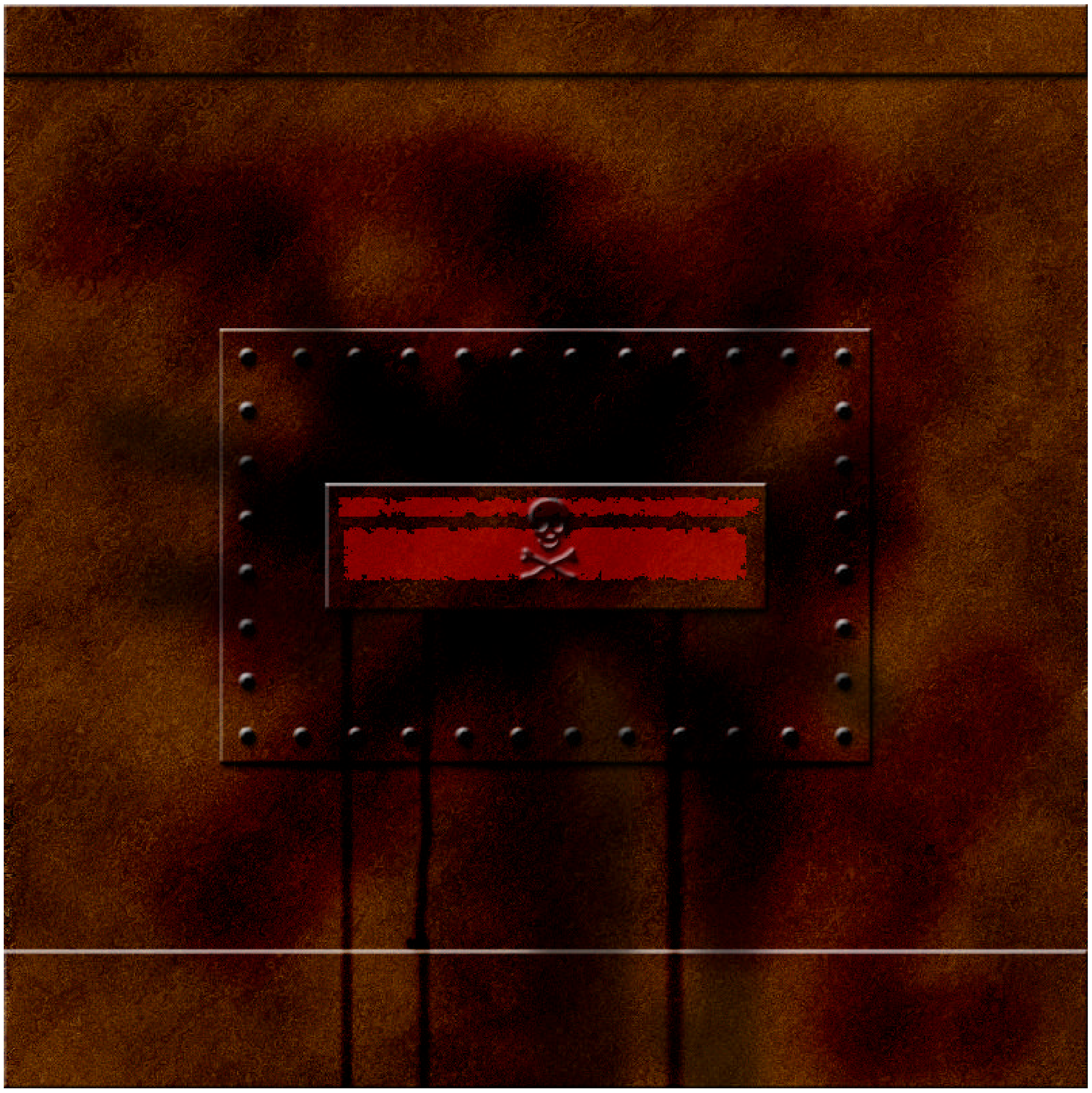


Figure 2.57 Here is the completed Space Base Wall texture with an ornament (the skull) and stains.

If you keep this image in layers, you can add or delete items like the vent cover and the burns. You can change the skull font with other fonts so the walls can indicate whether a player in your game world is in a good place or a bad place. Working from a layered image like this also helps keep the appearance of the game world consistent.

Moving On

In this chapter we learned the fundamentals of texture creation. We learned how to decipher what a base texture set is and then how to create one. We looked at the challenge of tiling textures and even how to do advanced tiling with natural patterns. Next, we will look at how you create more advanced textures and texture sets, using even more cool effects and techniques. We will also look at how massive texture sets are managed by game development teams.

Chapter 3
Advanced Texture Creation and Management

Once you have a set of base textures, you can begin creating detailed textures like those in commercial games and interactive products. Two things happen; you will create more textures than you dreamed possible, and you will have to organize them so they'll be of use to you and the development team.

Planning Your Texture Library

When you get your texture skills to an advanced stage, you will also need to bring your organizational skills up to the same level, or you will have a hard time finding those great textures you created. Before we look at creating advanced textures, we will look at the process of breaking out the texture library into a structure that will allow you to quickly create, save, and retrieve those images. Taking the time to properly break out and organize your texture library will also help you in the process of creating textures during development. The heads and subheads of a well-organized texture library can be used as a worksheet during development to determine the textures needed in any given area of your game. As you will see later in this chapter, a complex game world can be broken down into sets and subsets of materials, and this structure allows you to quickly go through the directory structure and use it as a checklist to review the textures needed in the world.

After we look at texture library organization, we will look at some advanced techniques for texture creation. You can go much farther than simply creating base surfaces like wood or stone. In this chapter, you will learn how to combine bases to make a floor tile that may be part wood and part stone, or a door that may be wood and various types of metal. You will learn about weathering, aging, and dirtying up your images so they not only look good but also blend together and appear to have been subject to the same environment. Weathering and aging also help fulfill the designer’s vision in the ‘setting and atmosphere’ department. For example, if you are creating textures for a game that features an ‘old and abandoned’ warehouse level, than you can develop the texture sets for the warehouse--crates, cinder block walls, and all the other surfaces you will need to cover your warehouse with--and then apply some weathering and aging that will make those surfaces look dirty, old, and even abused.

First, let's look at how to determine the textures you may need in your texture library and how best to organize them.

What Is a Texture Library?

A texture library is exactly what it sounds like, a library of textures. As with any project with a great deal of information involved, you will have to develop ways to organize and maintain that information. And as with books in a library, you will organize your textures in groups and subgroups, going down to any level of complexity that you need to keep track of the images in your library. This structure doesn't have to be terribly complex, but it does have to be usable. A simple system of headings, subheads, and a naming convention can fill your needs.

In addition to creating texture libraries (or the library of complete game-ready textures), you will also be managing many other libraries of resources, such as raw digital images, scans, 3D models, research, font collections, and so on. For example, I have the following libraries that I must maintain for the current project I am working on.

- **Raw Digital Imagery** - Since the project I am currently working on is tasked with re-creating reality, we (the other developers and I) have a massive library that contains thousands of digital shots of real world locations. We have long shots of places, medium shots, and hundreds of close-ups of surfaces and objects for texture creation, such as grass, stones, concrete, wood, tree bark, and signs, to name a few. We also have a huge library of high-resolution digital images of almost every weapon, vehicle, and object found on an army base. We are constantly adding to this library, and all these images must be named and stored in a series of folders and subfolders. We may have the following folder structure, for example.

  C:\Raw_Images\Weapons\MK_19

  In the 'MK_19' folder would be all the images of that weapon, usually 6 to 14 close-up views depending on the weapons' size. There are also images of the weapon being used, carried, fired, whatever we can get. As another example, we have the 'C:\Raw_Images\Signs' folder, where there are Traffic Signs, Warning Signs, Official Signs, etc.

  Note: The raw images are *never* deleted or overwritten; instead, copies are made to be worked on so we can always go back to the original.

- **Textures, Working** - These are textures that are in some state of completion between just started and almost complete. This library will almost certainly be the largest in both number of files and file size. As an example, in the Raw Imagery Library, I may have one high-resolution 2MB image of a wooden fence, but in the working directory I will have multiple versions of that wooden fence texture. In the game, the fence may need to cover a large area, but there will be areas where the fence may look different; it may be broken, faded, or burnt. While in the process of creating the fence texture, I will be working on unflattened Photoshop files in high resolution so these images will get as large as 50 megabytes or larger.

  Note: Photoshop can save images in layers, which allows you to edit and alter one layer without effecting the others. Working in layers is like having many clear sheets of plastic to draw on. On the bottom layer you may draw a grassy field, on the next layer you may draw a tree, on the next a cow — you can create as many layers as you like. If you don't like the cow you can move it or replace it without effecting the background layer. The downside is that each layer makes the file size bigger.

Usually I will create an initial version that is clean and fairly plain and that tiles well, and I'll save it (that's one 50 megabyte file with more to come). If I ever need that base tiling wood texture, it is there. I don't have to go through the steps of re-creating that texture in the future. Then I make a copy, and, with this as a base, I will create many other versions of the fence, versions that blend seamlessly as they are all created from the same base texture of the fence. As I create those other versions, they will each be stored as a high-resolution, unflattened Photoshop file.

It is a good idea to store many such versions of your images, in as many stages, as you possibly can provided that you have the storage capacity. It will be common to revisit textures and even entire texture sets to make changes, adjustments and new versions. The better your assets are organized, the quicker you will be able to fill texture needs.

* **Textures, Complete** - When a texture is finally looking its best and I am ready to put it in the game, I make a copy of it so the original (large unflattened file) will remain as it is. I take the copy and flatten it, reduce it in size and color depth, and perform whatever other operations need to be done in order to make it ready for use by the game editor. Remember, we are working large (I usually work at 2000 x 2000 pixels) and then reducing down to 256 x 256 or 512 x 512 -- a significant reduction in size. I then store all these completed game-ready textures as they are *before* they are imported or converted into the game editor. When a texture file is used in a game editor, it can sometimes become corrupted, or the texture library needs to be rebuilt quickly or reorganized for some reason, and you will be ready to quickly make those changes with the images all stored this way. Having the textures stored this way also is handy if you want to print out contact sheets or show the textures in the game by using Photoshop's Contact Sheet function.

* **Texture Libraries, Game Formatted** - Finally, you will most likely end up with texture files that are in some unique form to be used by the particular game engine you are working with. In some cases the game engine may read textures from directories in their native format, reading common file types such as bitmaps directly off the hard drive, not dealing with a special format, but usually you must import textures into a special file format for the game engine to read. Most world editors require that you import textures into a library that their software maintains because there are special variables that can be assigned to textures and special effects. Also the texture import process can validate files by checking that they are the right size, dimension, color depth and other requirements. Once you import the image, you can't get to it using Photoshop; you access it via the editor for use

in the game world.

Texture libraries should all follow the same organizational structure as closely as possible. If you have a raw image stored in C:\ground\grass\dead_grass.jpg, then you should also find in the 'textures, working library' a folder called "grounds" and a subfolder named "grass" with many versions of dead grass. In the texture libraries used by the world editor, you should find the same structure as well. The confusion of having a different structure or naming convention for each step of the process will quickly become overwhelmingly confusing, if not paralyzing.

A consistent naming convention and organizational structure also makes it much easier to generate a status report on texture progress based on what library the texture is in. You may have a simple worksheet, something like this, for a texture artist to maintain.

Texture Status Report

Level: The Old Warehouse

Artist: (Artist Name)

W = Working, R = Raw Image, C = Complete

@@@Production: The following minitable is part of this sidebar. @@@

Texture name:	Currently in:	% complete	Filename
Ground, Gravel	W	75	gravel001
Ground, Gravel w/grass	R	0	gravel002
Ground, Grass, dead	C	100	grass004

@@@end minitable@@@

These are the libraries of assets I have to deal with on this particular project, but if you were making a game that used no digital imagery and you were not trying to re-create the real world, then you might not have the Raw Image Library. In fact, you may have only the working and the complete stages. Some games may rely on 3D art and will therefore have 3D models, their textures, and final renders of the 3D scenes.

Those are the basics on how to organize a great many game textures and assets into libraries, but before you even begin to create those textures, you must know what textures you are creating. To determine this there are a few questions you must answer first.

What Textures Need To Be Created?

Now that you know what a texture library is, you need to figure out what textures need to be created to fill those libraries. And you probably won't be doing this by yourself. Often it is not only the texture artists' job to help break out the texture needs of a game; rather, there's a meeting of the minds with all the individuals on the team who are trained and experienced with the game-type being designed and the technology being used in development. The more people who are involved, the more factors you might have to consider.

When the designers, producers, level designers, and even programmers all see what you have initially broken out as a list of textures needed for the game, they will most certainly have feedback for you. You may have broken out a texture list that contains sky boxes, or a set of six images that seamlessly fit together to fill the inside of a cube that will form the inside of your world’s sky. The programmer may come to you and explain that a new process for the sky is being used that does not require the tedious ‘sky box’ technique but involves flat images that will appear to go off into the horizon as a real sky does. The level designers may come to you and explain that they are using a new approach to building sand and they need more variations on the base building materials than you have listed — and on and on. You’ll need to keep track of all these requirements, and then some. So in your preparations to break out your game’s texture needs, here are some questions that need to be answered.

What View of the World Does the Game Offer?

The first consideration when building a texture library is the view of the world the player will have. The player’s POV (point of view) will determine a great deal pertaining to your texture needs, including how much detail you need to put into the textures and even what your approach will be in creating the textures. A door seen from a car driving by at 200 miles an hour (when you are the driver) doesn’t need to be at the same level of detail as a door that is seen from the player’s eyes when that player is able to walk right up to the door and inspect it. The ‘200 MPH drive by’ can be a small brown square, whereas the face-to-face version of the door will need more detail and also need to be a larger image. Building textures for a first-person shooter will require a very different library than that of a third-person game, a god game, or a racing game. So the genre influences or even determines the POV.

The type, or genre, of a game is the most fundamental bit of information you must first gather. Let’s look at the types of games on the shelves and the texture needs for those types of games.

* **First-Person View**. Usually this is a close-up view that shows the player detailed doors, walls, and interior spaces. If you are able, open the texture library of a typical 3D shooter, and you will see many walls, floors, and ceilings as in figure 3.1.

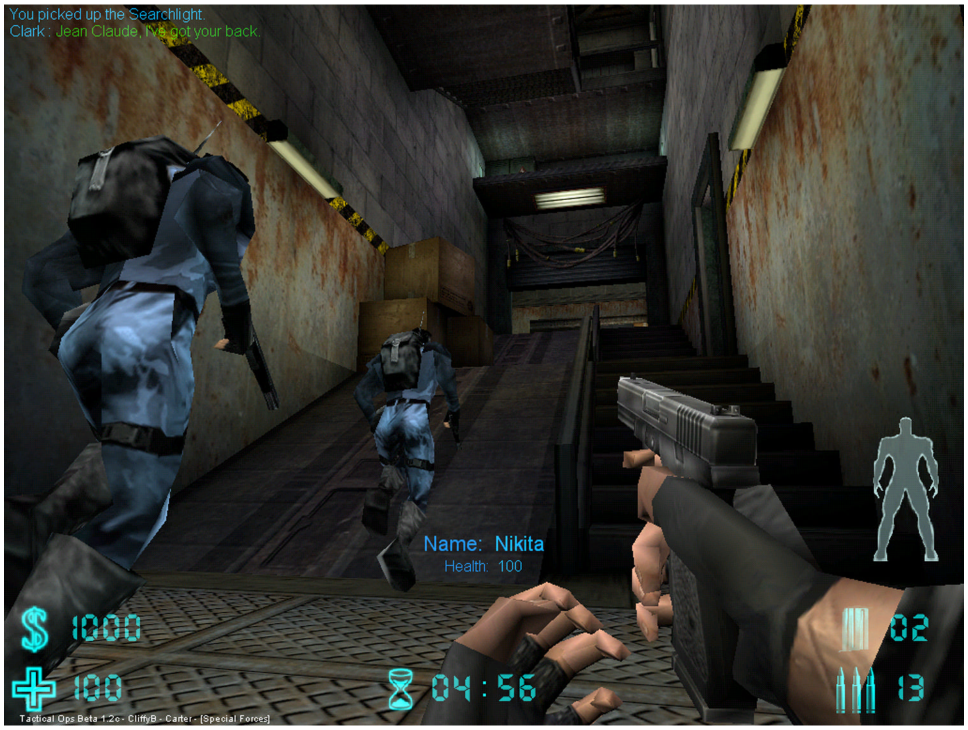

You picked up the Searchlight.
Clark : Jean Claude, I've got your back.
Name: Nikita
Health: 100
1000
100
04 : 56
02
13
Tactical Ops Beta 1.2c - CliffyB - Carter - [Special Forces]

Figure 3.1 This is the first-person view of a game world. The level of detail for this view must be higher than for some other views.

* **Third-Person View**. Similar to First person, but the camera has a broader range of movement. Often, you, as texture artist, must be concerned with the first-person view as well as with a broader picture because the camera will often give the player visual access to portions of the game not seen in first person as in figure 3.2.

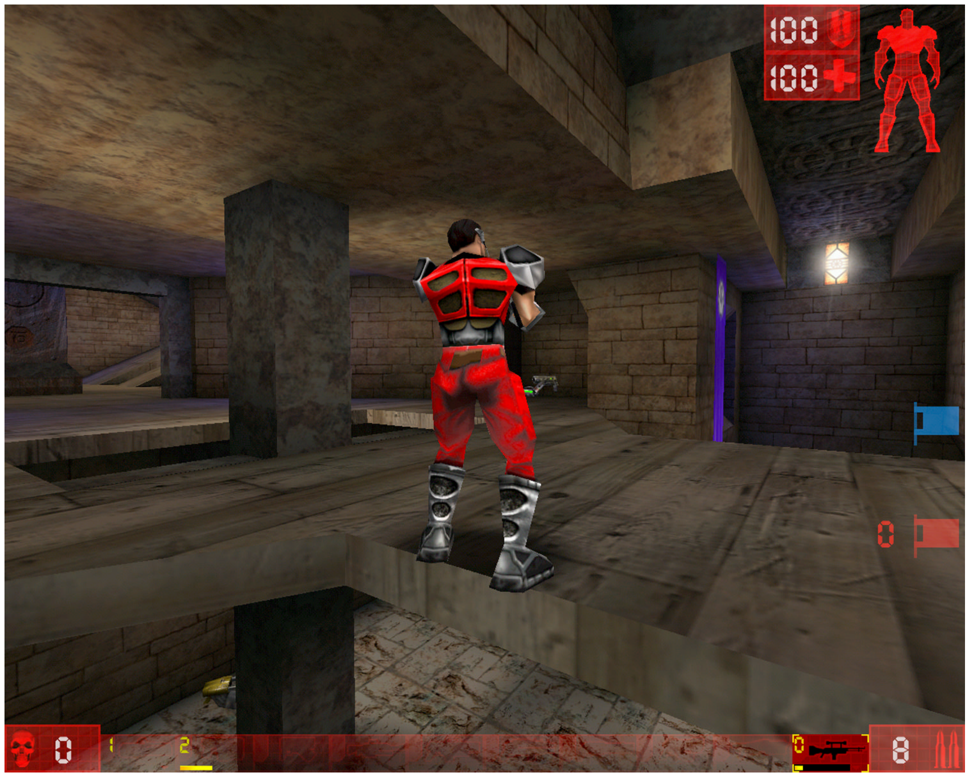


Figure 3.2 This is the third person view and this view often lets the player see parts of the world that the first person view may not let you see.

* **Vehicle-Based View**. This can be like first person if the vehicle is in a building, but many vehicle-based games take place in airplanes, tanks, racecars and space ships. The texture needs of a racing game will be different from those of a tank game. Tanks are generally allowed to slowly roam a large piece of terrain (grass, trees, sky), whereas a racecar generally streaks along a predetermined course (road, banners on the stands, sky). As you can see in figure 3.3, a helicopter offers an even larger view area.

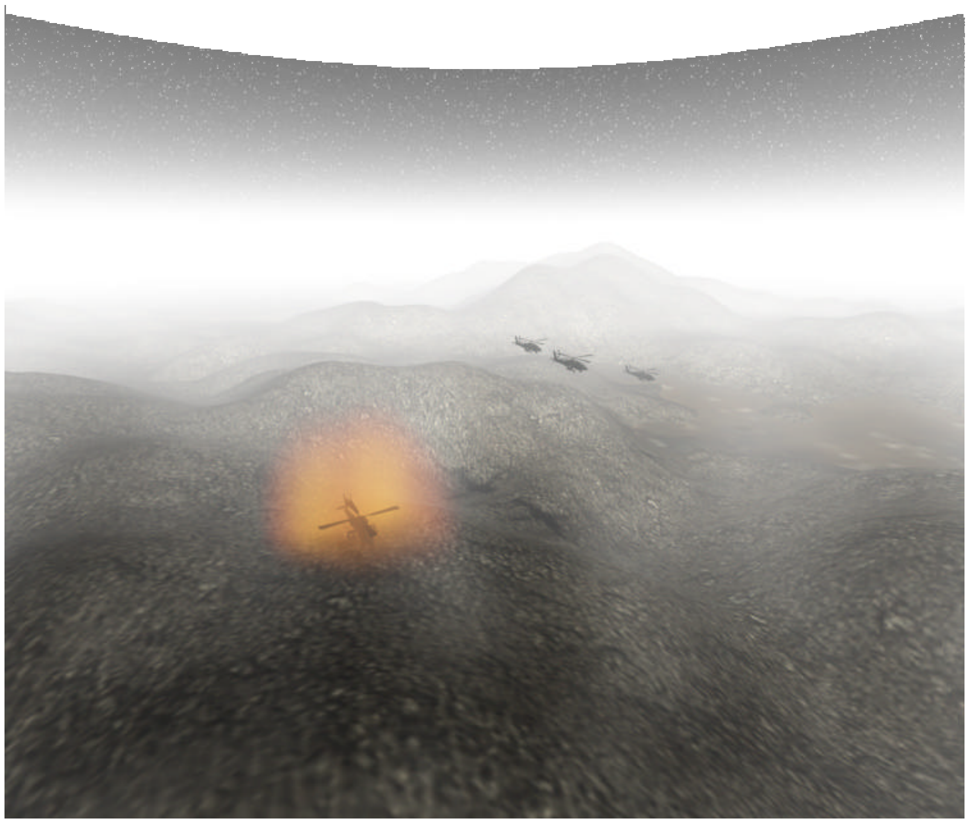


Figure 3.3 This is the vehicle based view. Of course flying a helicopter lets you see more of the world.

* **Orthographic View**. This view looks down on the game world and usually offers a broader view where up-close detail is not as important as seeing the layout of the surrounding streets or hallways of the world as in figure 3.4.

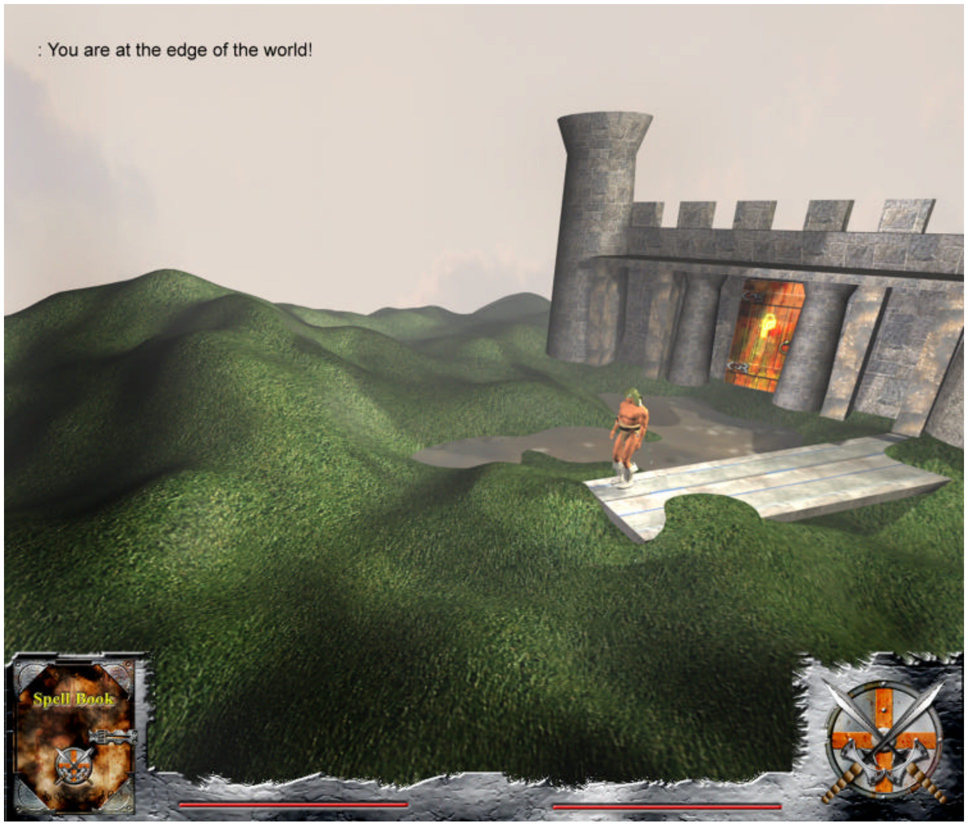


Figure 3.4 This is the Orthographic view, a broader view than third person, but not quite as big as a helicopter.

As you can imagine, combining any two or more of these game views will at the least add to your work load, if not exponentially multiply it. Imagine a game that allowed a player to go from a first-person walkabout to flying a helicopter. There are games that allow this, but the tradeoff is simply that the world is large and not very detailed. And the artists, programmers, and all involved had to do additional work in every area to create this functionality. New code had to be written, design had to accommodate the expanded player world, more art had to be created (or used more repeatedly), and more sounds needed to be generated, models created, etc.

Where and When Is the Game Taking Place?

Game setting and atmosphere are very important in the equation of generating game assets. If you are building a game that is set in a city of an ancient civilization as opposed to one set in an open terrain setting where there are only a few modern work sheds, then your texture needs and focus will be very different. For the ancient city, you may be creating hundreds of textures for buildings, statues, ornamentation, and more. For the modern world game, set in vast terrain with a few simple buildings, chances are you will be focused on creating many ground textures and spending a great deal of time tiling and tweaking them because they are visually prevalent in the game.

Game setting also affects the organization of the texture library. In the case of the ancient city, with all of its marbles and ornamental pieces, you may have several libraries just for marble textures. In contrast, the game that has only a few marble textures show up anywhere in the game world may have that texture labeled under another library of materials or even ‘miscellaneous’.

Finally, game setting will also greatly determine how you make your textures. For example, in one game development project, we had to make every single texture in a 3D application because the game was set in outer space on alien worlds where literally nothing we were creating existed anywhere in this world. We spent hours planning and experimenting with 3D modeling, lighting, and rendering. We were indoors almost all of the time during development in high creative mode; we were always brainstorming new worlds to create. On the other hand, I am currently involved in a game project that is quite the opposite. Every single object and surface exists in this world and must be accurate. We spend a great deal of time literally in the field with digital video and still cameras capturing every detail possible. We also spend a great deal of time learning reality (not making it up) and trying to get that into the game. We rarely use a 3D program (for textures), and we use Photoshop a great deal to touch up, clean, and manipulate digital images. So for different game settings, you might be working indoors or out, being freely inventive or painstakingly realistic, creating images or scanning and fixing them.

How Is the Game World Divided?

What are the common game world divisions? This information partially comes from game type. If you are developing a driving game, an indoor shooter, or a massive outdoor level, your game world divisions may be floors, ceilings, and walls, or tracks and banners, or in the case of a large outdoor level, they will be forest, desert, and polar ice. How the world is divided obviously determines what the focus of the game will be. A shooter that is mostly indoors will focus on making the up-close details of floor, walls, and indoor ornament look good, while the massive outdoor game will want the grounds to look their best; grounds should tile well and look as good as possible since they are the main focus. There may be more sky textures to deal with, trees and plant life, and even weather effects.

How the world is divided may break out texture libraries like so:

3D Shooter (primarily indoors)

- Walls
    - Brick
    - Concrete
    - Wood
- Floor
    - Tile

        - Metal plate
        - Wood
    - Windows
        - Panes
        - Sills
        - Glass

Outdoors Game

- Grounds
    - Dirt
    - Grass
    - Grass, dead
    - Mud
- Trees
    - Leaves
    - Bark
    - Branches
- Water
    - Blue
    - Dark
    - Dirty

What Technology Will Be Used to Develop the Game?

How the game world is divided from a texturing and assets point of view is greatly derived from what game technology is being used. If you are using all your computing power to render large and ornate rooms, then chances are you can’t also have vast and impressive terrain. And likewise, a game with vast and impressive landscapes will have small and simple buildings. Even when we have games with both, we will still be trading off, meaning that a game with both a large terrain and detailed buildings could lose the buildings and have even larger terrain, and vice versa. Just as there are many game types, there are many game engine types that each has a type of game they process best. For a good article explaining all this go to www.3dactionplanet.com/features/articles/gameengines.

Technology also determines organization. In the Unreal editor, we are able to create texture libraries and subgroups in those libraries. This makes organization easier and makes communication between artists and level designers easier. With Genesis3D, which you will learn about in part2, most of your organization must be done outside the editor and the texture library built before you open the editor. With Unreal you can import images as you work and organize them in the editor.

Organizing the Texture Library

Once you have the texture requirements for the game, you must go about organizing them into a library that will be easy for you, and any other artist working on the textures, to access and update. You will also have the level builders using your textures, and the level builders need to know where to find what they need. Here I am strictly addressing the organization of assets by groups and names. Another aspect of having many people using and working on the same assets is file management, where special software like the software programmers use for version control is used to make sure that no file is deleted, corrupted, or overwritten. We will not touch on that here, but it is a topic you need to address in game development.

There are more than world textures in a game (floors, walls, and ceilings); there are also textures for the HUD (heads-up display figure 3.5), splash screens, menus, and many other places where art is used. The asset list below is for a typical 3D shooter, and as usual, your needs may add, subtract or break out any of the below headings. A game that is strictly indoors, or has few trees, may not have an entire library devoted to trees. A game with a few trees may have those trees stored under “organic,” “miscellaneous,” or even as a subgroup of “wood.”

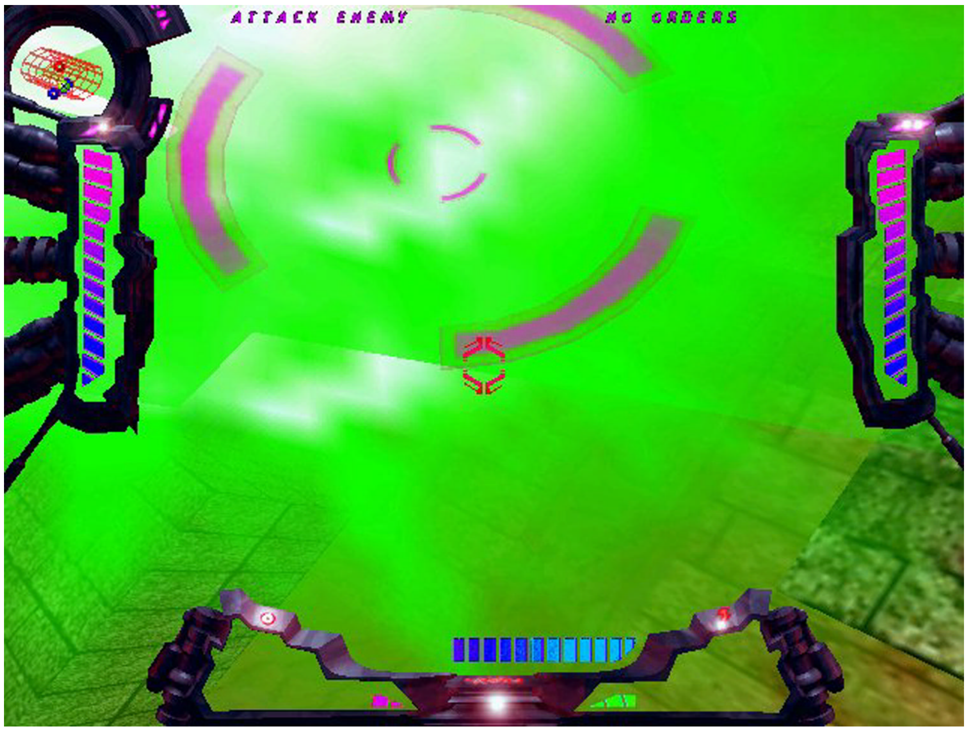


Figure 3.5 The HUD or Heads Up Display is the interface overlying the game where the player can see their health, armor, ammunition, and other vital information.

Table 3.1 Texture list for a typical 3D shooter.

Asset	Description
Buildings	Building textures usually tile in one direction when they are walls with a notion of top and bottom. Some games need walls to contain details such as pipes and windows; whereas other games will model the pipes and windows separately and require only a tiling texture of a brick or stone. Building textures are images such as roof tiles, trim, wall surfaces, windows, floors, and doors.
Decal	Decal textures are small textures with an alpha channel (an invisible area of the image) used for stains and marks on the game world. Decals are also used for small posters, signs, and even light switches on walls.
F/x	These are textures created for use during the game and not put in the level by a level editor, they are usually handled by the programmers and called by the code during game play. These are the textures used when a dynamic lightning bolt strikes, when a weapon is fired and you see the muzzle blast, or when a bullet hits the wall and you see the small bits fly. These textures are also used for in-game effects such as smoke, explosions, fire, sparks and other particle type systems.
Masonry	Masonry includes tiling bricks, cinder blocks, concrete, stone, and heavy surfaces such as that.
Metal	Metals can be tiling sheets of aluminum, corrugated tin, beams, pipes and more.
Missions	Sometimes you may have images that fit no real category but they appear in a level so they need a home. I don’t often find these cases but usually we reserve this library and when we find that we are sticking images in it we examine the images and it usually turns out that we need to create a new library for them.
Organic	Organic textures are different from terrain textures (below) because organic textures don’t have to cover large patches of terrain. They are designed to cover smaller areas and objects such as rocks, plants and dirt patches in the terrain.
Signs	In any real life situation, or places inhabited by civilization, you have signs. Traffic signs, warning signs, official signs, business signs, and the many signs we see everyday.
Terrain	These are maps that are to be used on large terrain areas, typically grass, rock, and dirt.
Trees	Trees can include bark, branches, leaves, and billboards of trees.
Wood	Wood includes beams, crate sides, paneling, plywood and usually manmade wooden items.

When you’re planning the texture needs, organization, and methods of texture and asset creation, keep in mind the player view, location and setting, world divisions, and game technology of the game. Any of these can drastically alter the example above. An indoor game may not have a terrain library, and many games have few to no signs and therefore will not require a library dedicated to signs.

Now that we can organize textures with the best of them, let's create some more textures. We will start by learning to combine the bases we made in the last chapter to create all new and more complex textures.

Creating Advanced Textures from Bases

If you look closely at the world around you, no matter where you are, you will see that it is composed of some basic surfaces. In my office there is carpet, white sheet rock walls, ceiling tiles, and the weird stuff our desks are made of — compressed meat loaf I think. These are all plain, easily created and easily tiled textures; see Figure 3.6. Figure 3.7 shows a scene using these four basic textures.

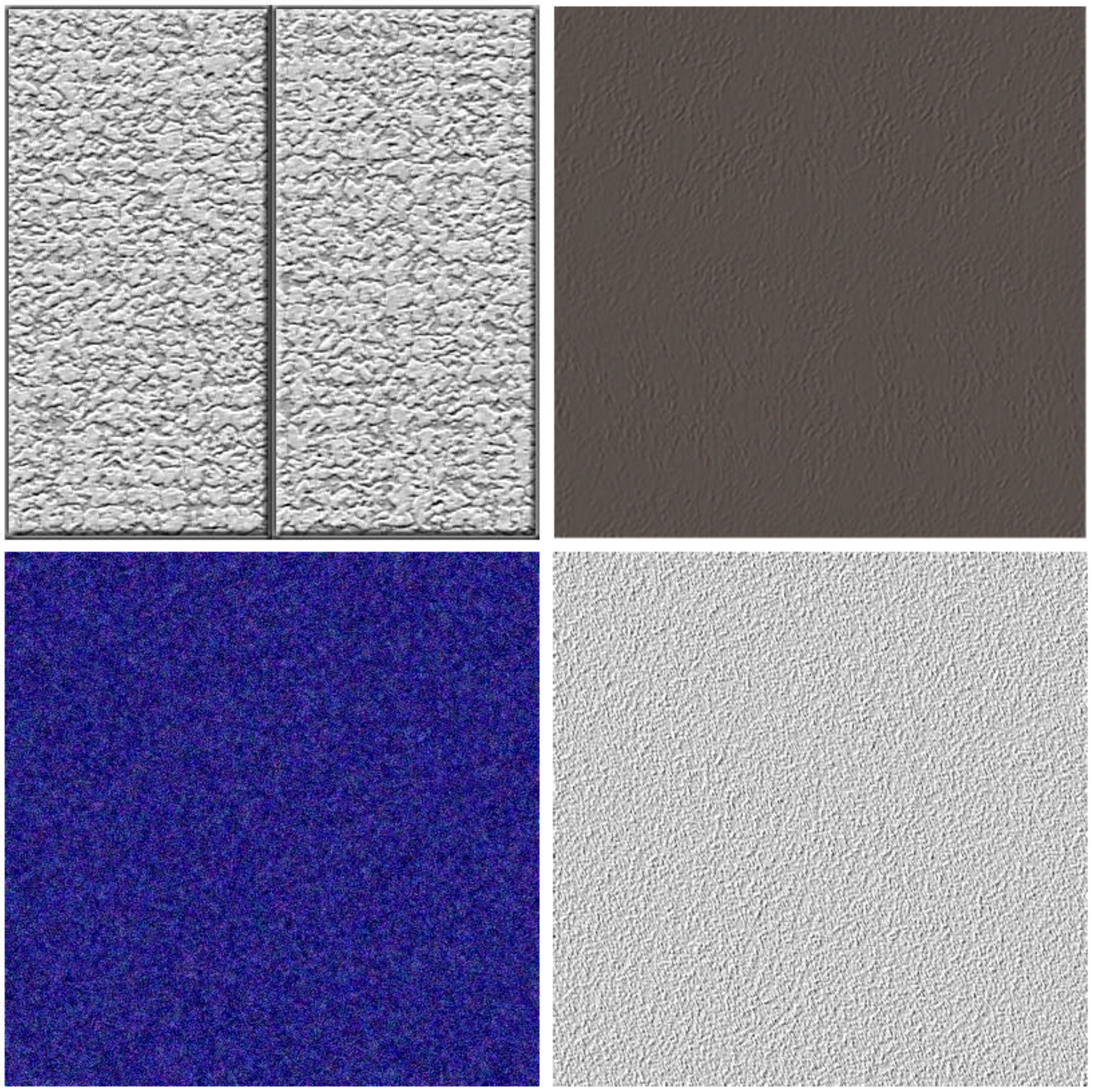


Figure 3.6 The four basic textures used in a modern office scene. These base textures are easy to create and tile.

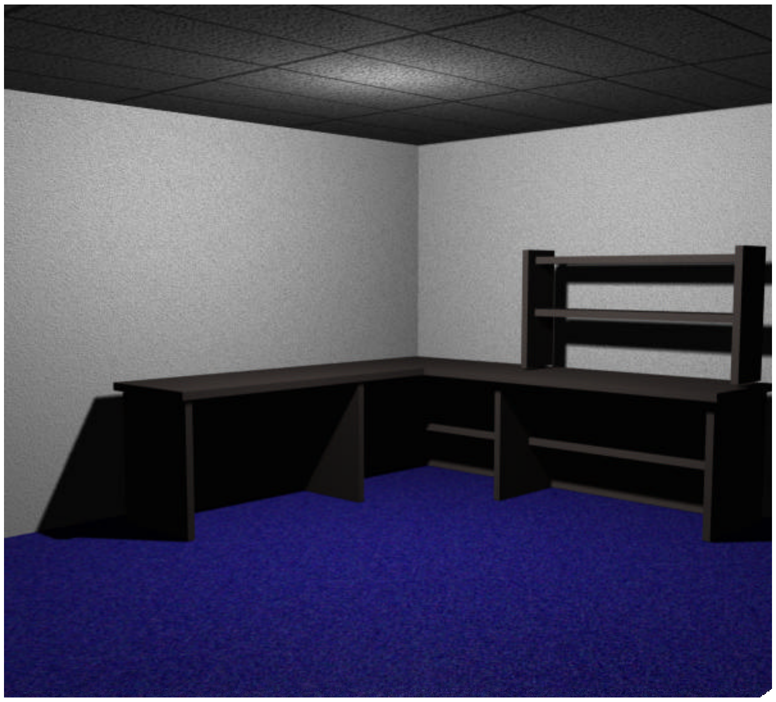


Figure 3.7 The office scene with textures applied, plain but pretty.

Other worlds are created in the same way as ours, with basic surfaces underneath as well. Consider a castle for example. You may have walls made of a base stone that is a bit more complex to come by and to get to tile just right; then you have wooden beams that hold the tiled slate roof up; and then you have the ornate wooden door with banded iron hinges and really cool metal studs, all dripping with rust and slime — and suddenly we have hit the point where a few tiling base textures don’t fill the bill anymore. Figure 3.8 shows the castle scene using only the base textures to cover the geometry.

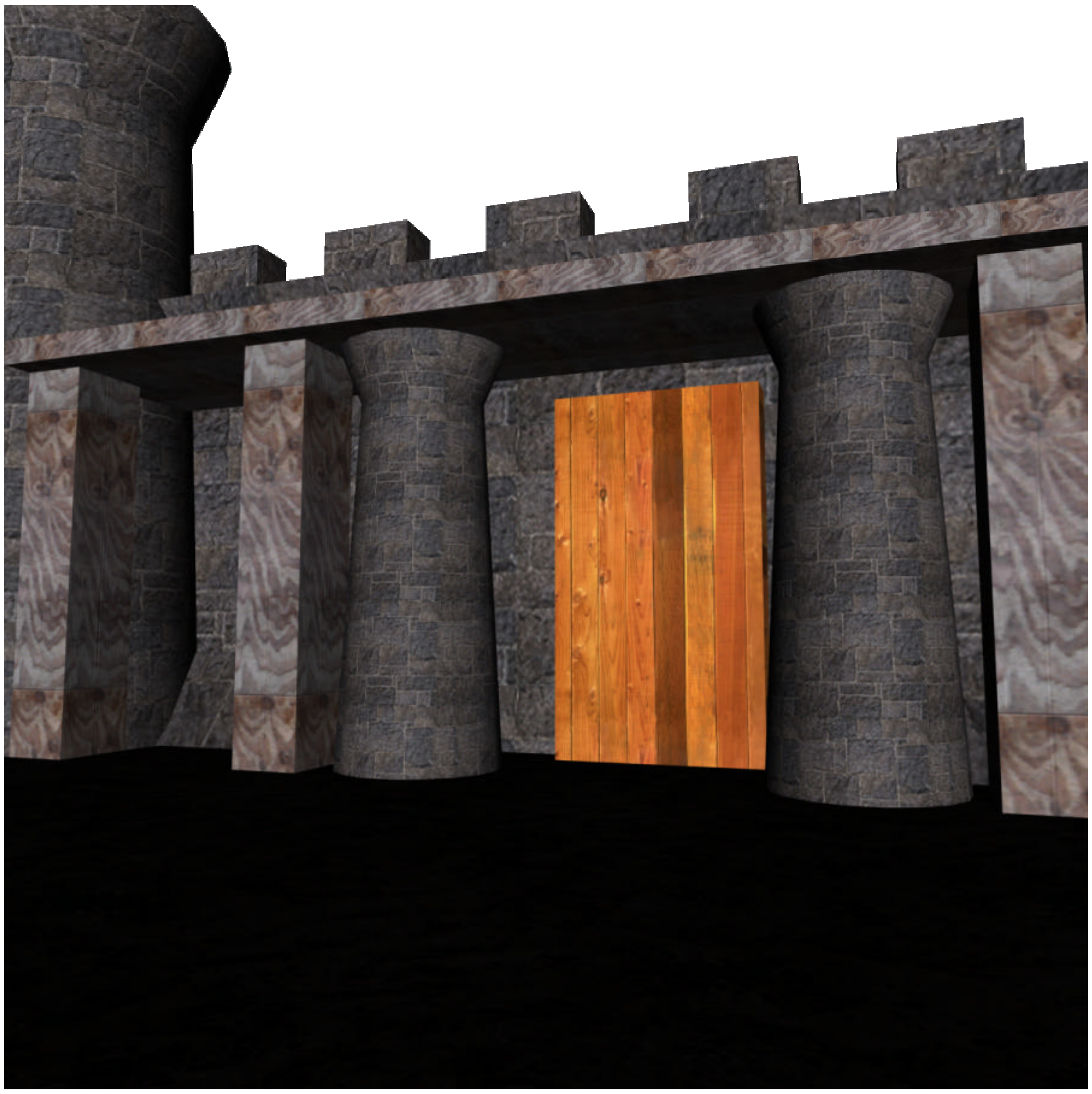


Figure 3.8 Here is a castle with brand-spanking new stone and a door made with wood, but where are the hinges, the knob, and all the — well, frankly, where is the believability in this scene?

While the office scene looks pretty good and believable (because it is an office and it is acceptable that the office may be smooth and have clean surfaces), the castle scene is lacking in a great deal of atmosphere and believability because it is so clean. Not to mention the castle door, which is simply several wooden slats at this point. With the addition of weathering, aging, and combining textures, we can really make this castle scene come to life, as in Figure 3.9.

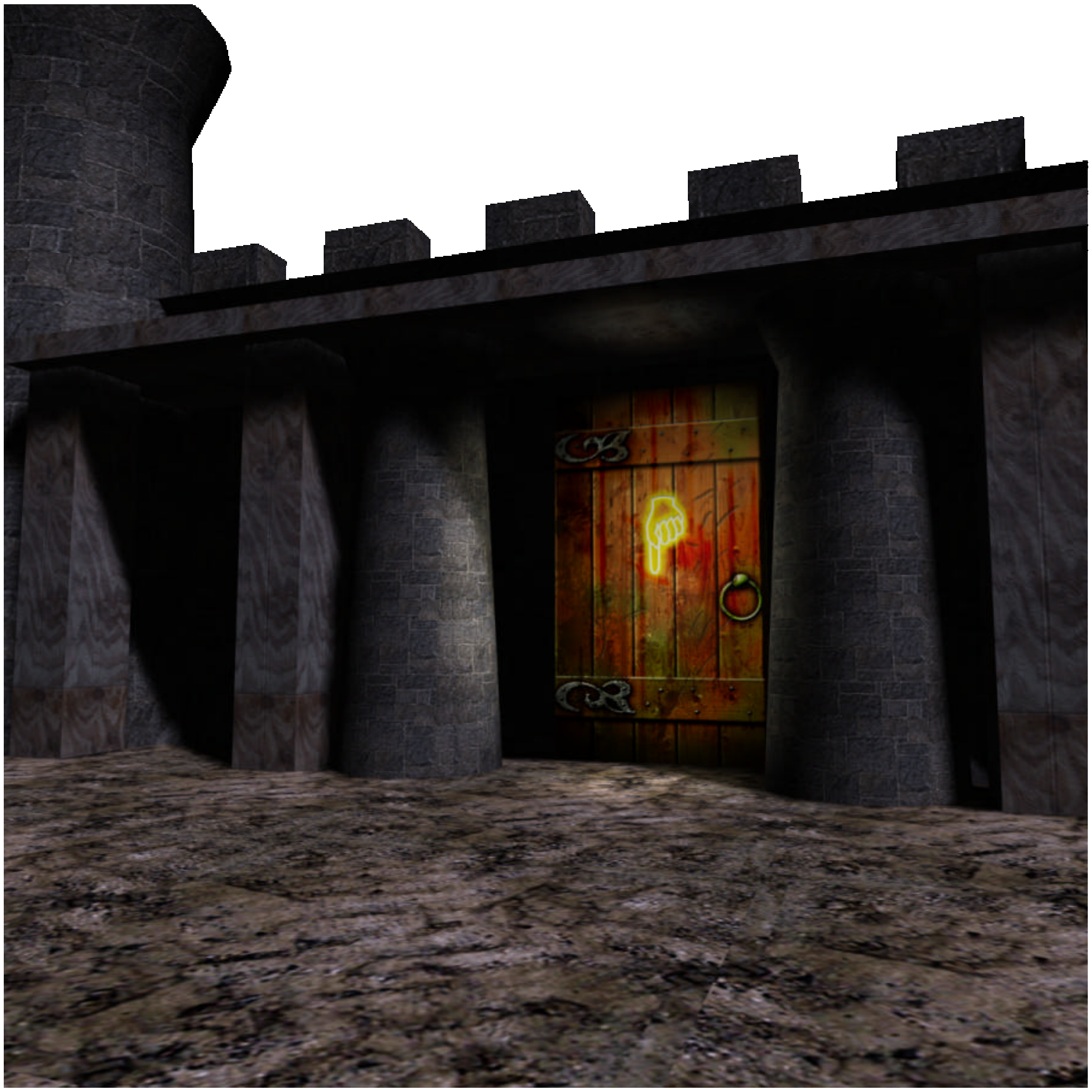


Figure 3.9 Here is the castle scene with the door texture. The glowing hand is magical symbol of protection added for effect. You will learn to make the door texture at the end of this chapter.

If you are tasked with having to create the textures for that castle scene and the door, complete with brass pull ring and ax chop marks, how do you start in creating that complex texture set? Simple, you break that texture down in the same manner as you have broken down the world space as illustrated in chapter 2. The castle door in figure 3.10 is composed of a base wooden board, a base metal texture, and that cool glowing rune. If you look closer, you will also notice that the door is not just those base materials laid on top of each other; there are also highlights, shadows, aging (weathering), and some wear and tear in the form of nicks, dents and stains. In this section you will learn to apply all those effects quickly and easily to your materials to create stunning textures.

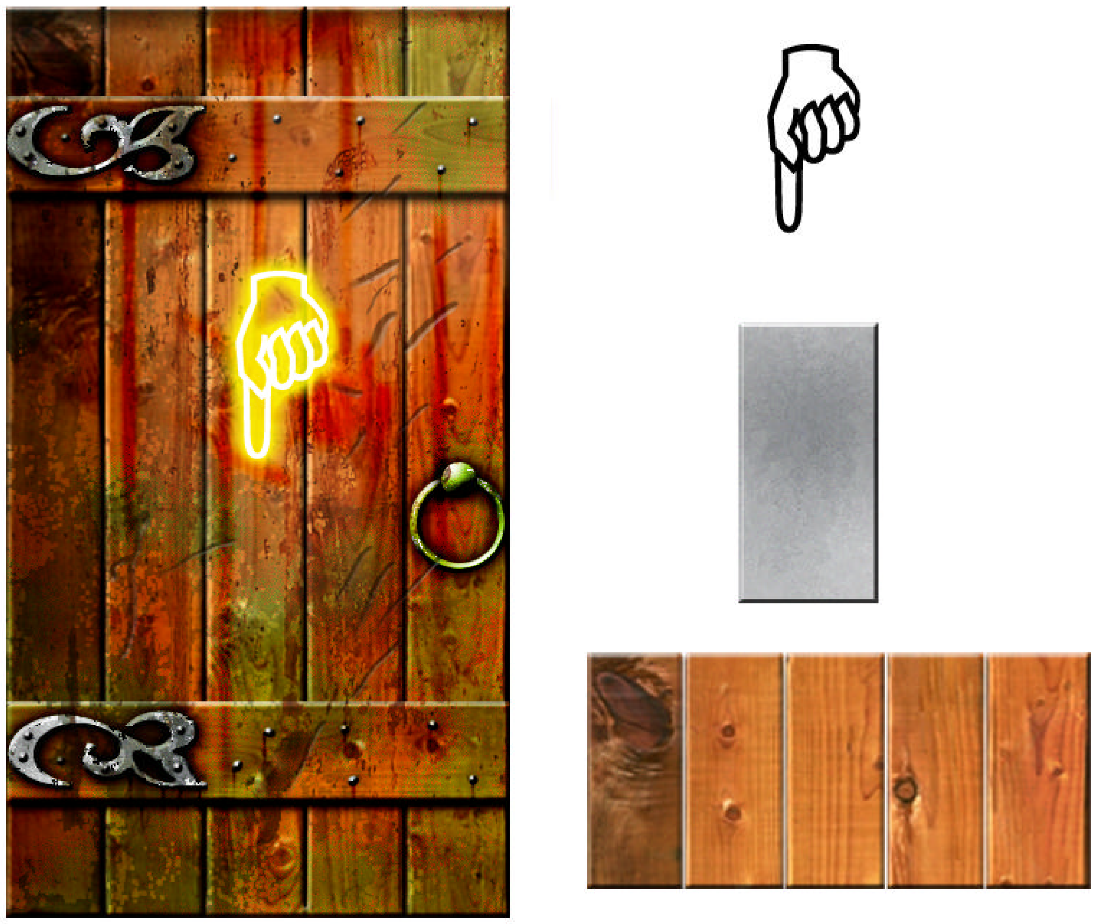


Figure 3.10 This intricate castle door can be broken down and re-created in minutes once you know how to pull out the base textures and then apply the basic lighting, aging, and other details to the texture.

Let’s start by learning how to break down our texture needs as we did in the previous chapter. In chapter 2 we looked at how to look at a scene and break out the base textures, or common tiling images, that cover the common areas of our world. Now we will look deeper at how we can break out detailed objects and surfaces in order to re-create them in Photoshop.

Combining Base Textures: the Importance of Highlights and Shadows

Some of the easiest and more impressive textures you can make from combining bases are ornate floor tiles. And as simple as a floor tile may be, they require the same attention to detail in order for them to look good. And some of the most important details you should always be paying attention to in your textures are *highlights* and *shadows*.

In order to give depth to a 2D image, we can easily add highlights and shadows using the 'Bevel and Emboss' layer effects in Photoshop. The trick to making a 2D image have depth and appear to have three dimensions is to know how far you can go with the effect. Some highlights and shadows are necessary for an image to have depth, but push the effect too far and it will actually have the opposite effect and make the image appear to be flat. Look at figure 3.11 and you will see that the combined base textures in this object look flat, but with some highlights and shadows it has some nice depth (as seen in figure 3.12).

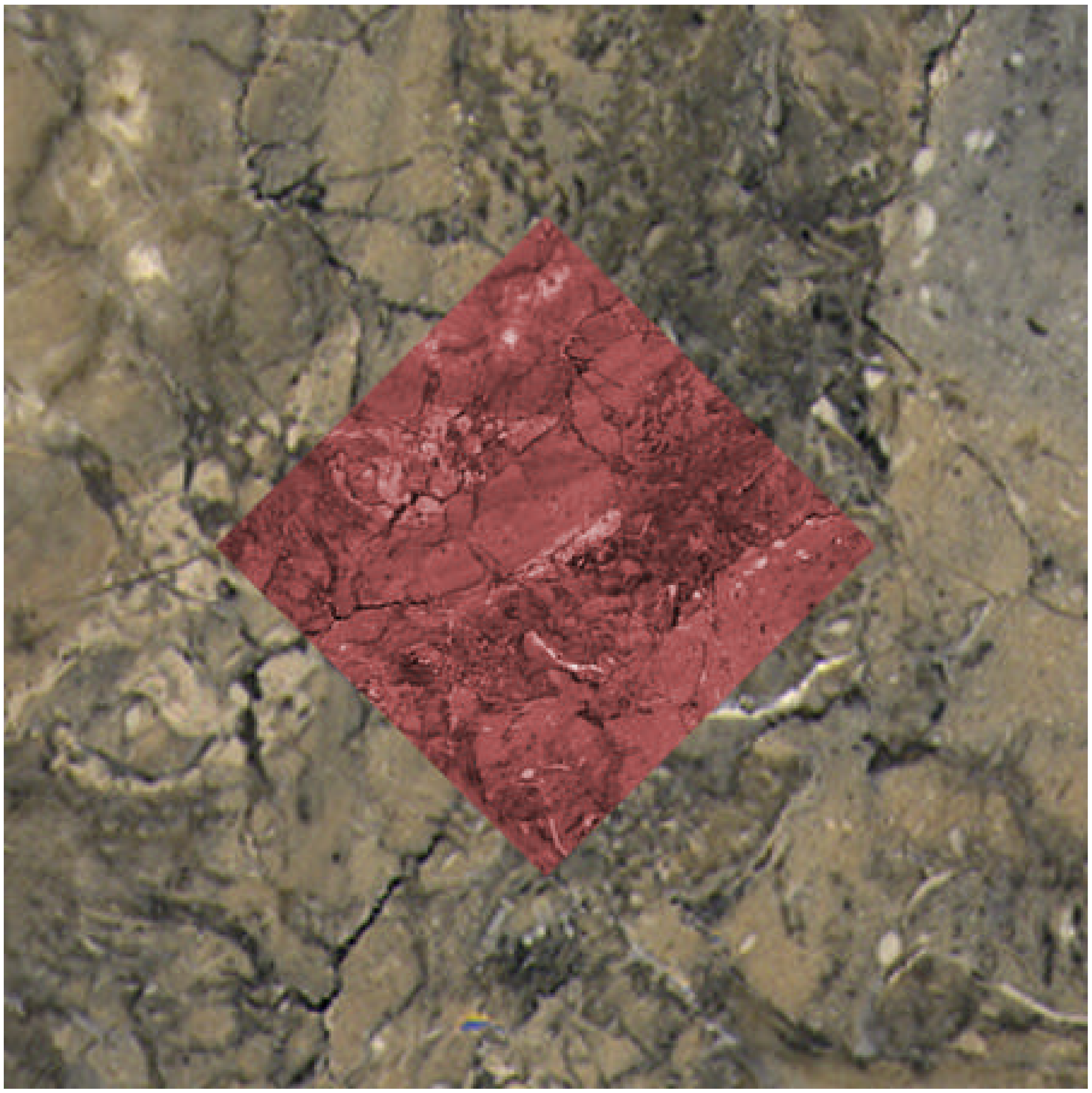


Figure 3.11 Combined base textures with no highlights or shadows are very flat.

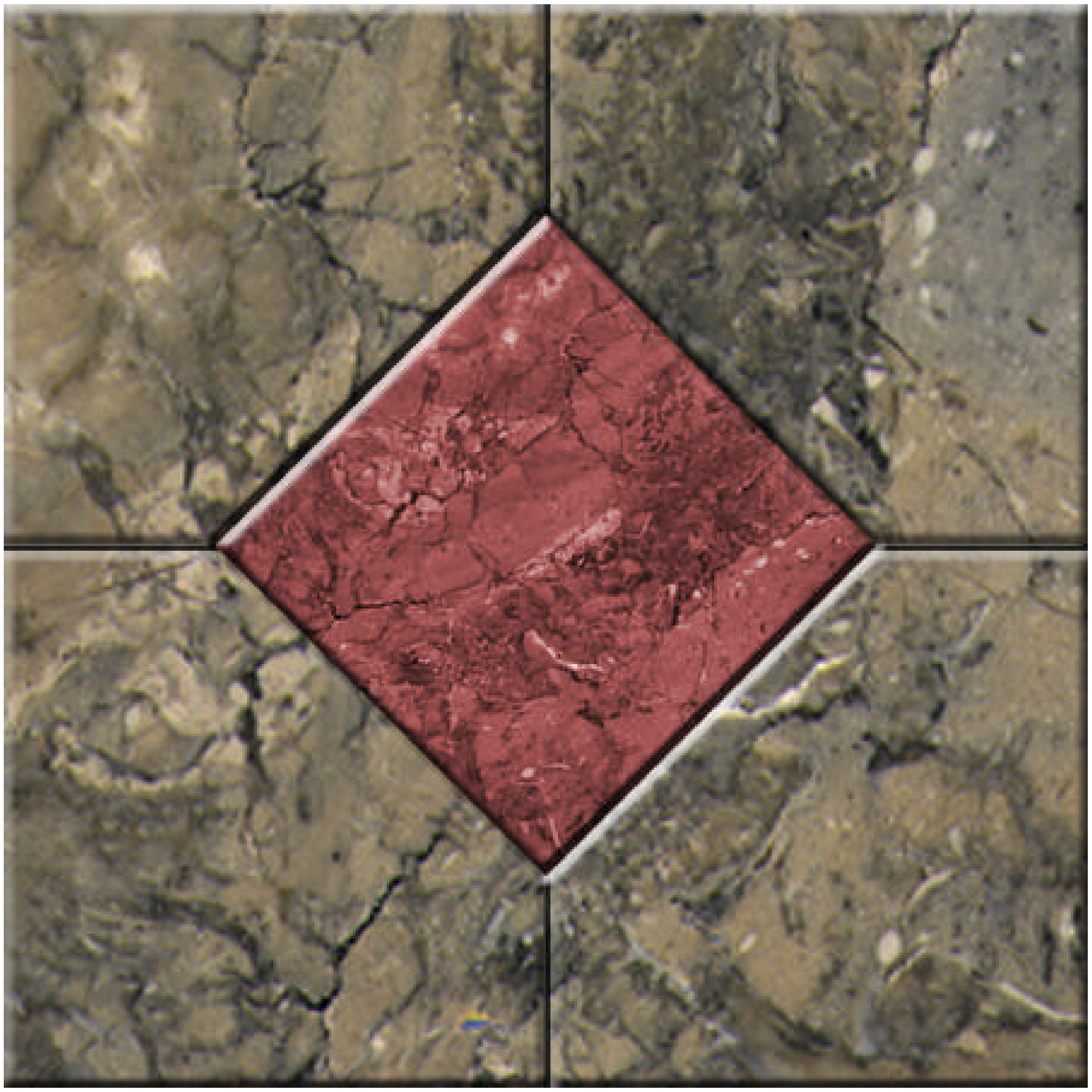


Figure 3.12 Add highlights and shadows to the base textures and you have a nice image.

Creating a Floor Tile

As we did in Chapter 2, we're going to work in Photoshop here. Start this exercise by opening the file 'marble texture' from the CD. If you look at the image of the completed tile, you will see that it is composed of nothing more than a copy of the initial marble texture layer that has been repositioned and colored to fit the tile pattern. In order to do this, do the following.

1. With the open "marble texture" file in Photoshop, copy the layer twice. You will not be using the very bottom (or background) layer, but since you can't apply effects to the background layer, you need to copy it for this exercise.

2. Color the topmost layer using Image | Adjust | Hue/Saturation. I simply clicked the Colorize option and it gave me the color I wanted, but you can play with the Hue, Saturation, and Lightness sliders to get any color you want.
3. Use the guides to divide the image perfectly in half horizontally and vertically. Optionally you can use the grid tool, but I find it easier when dividing an image in half to go to the background layer and drag out the guides until they snap in place.
4. Make a square selection on the top layer you colorized (holding down the Shift key so the selection will be perfectly square), and center the selection by snapping it to the guides. Right mouse click on the selection and Invert the selection by choosing 'Select Invert' from the context menu.
5. Delete the selected area of the image. You should now have a small square of colored marble in the exact center of your image. Press Shift + 'T' to Free Transform this layer, and hold down shift as you do so you can rotate the square of marble exactly. Your image should now look like figure 3.11 above.

Now let's make the floor tile look like figure 3.12.

1. Start on the first layer above the background layer and apply the Bevel and Emboss filter. Use Inner Bevel and leave the default settings.
2. Copy the effects by right-clicking on the layer and choosing the Copy Effects option. Then paste the effects into the top layer where the smaller marble square is.
3. You may notice that figure 3.12 also has the lines that come from the corners of the small marble square. To accomplish that effect, simply create another layer between the two marble layers, paste the effects into it, and then draw some 3-pixel-wide lines in. Holding shift will make the lines straight and using the guidelines will make them straight.

We made that floor tile image pretty quickly and it looks good. Here it is in use in figure 3.13. Next we will make a more complex object from the ground up.

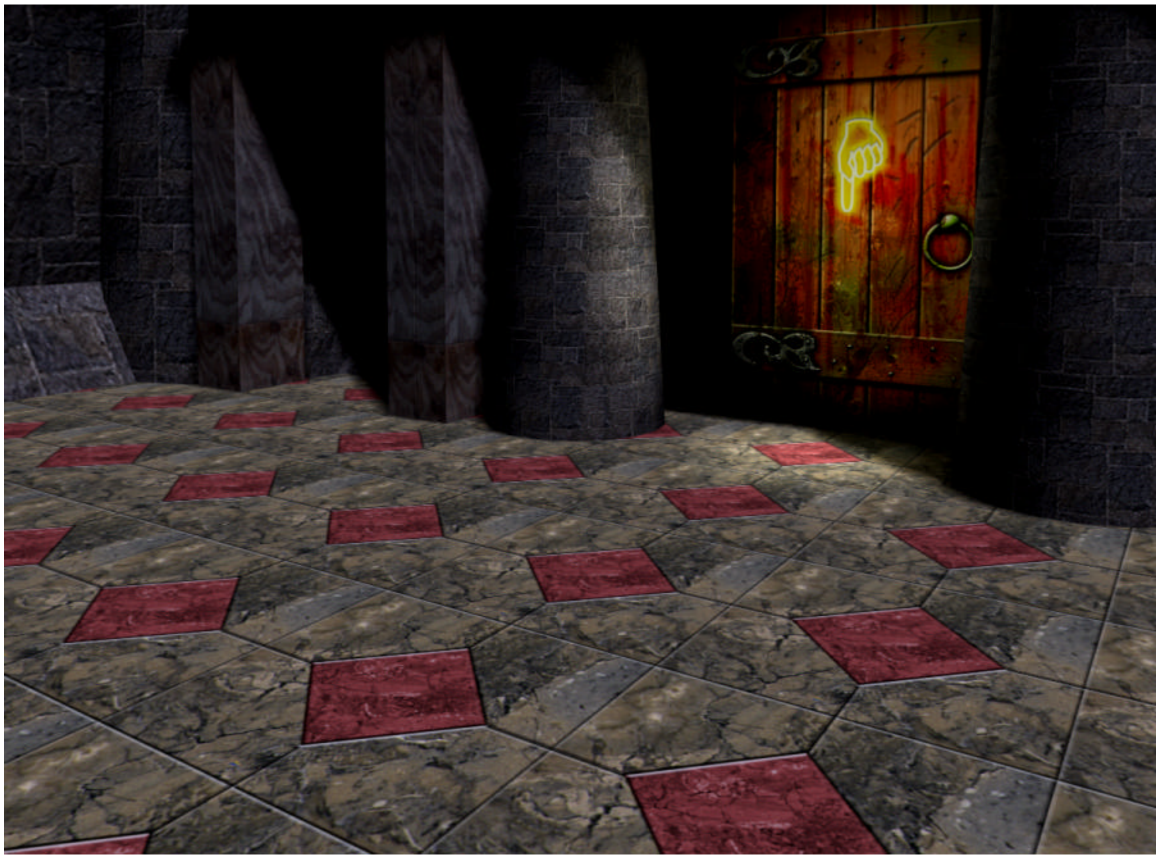


Figure 3.13 The floor tile created in minutes in use in a scene.

Making Complex Objects and Dirtying Them Up

This is the tale of how one day I had to build army-issue ammo boxes from nothing. Yep, we needed an ammo box texture and there was little time to make it, so I went on the Internet and typed in Ammo Box and built my texture from scratch. I simply started with an army green layer, made it dirty, cut out the areas that needed depth, and I was done.

Building a complex object is that simple, but I will walk you through the steps here. Building complex objects is easy because when you break them down, they are never as complex as they seem. The ammo box is a great example. In figure 3.14 is the actual photo reference I started with.

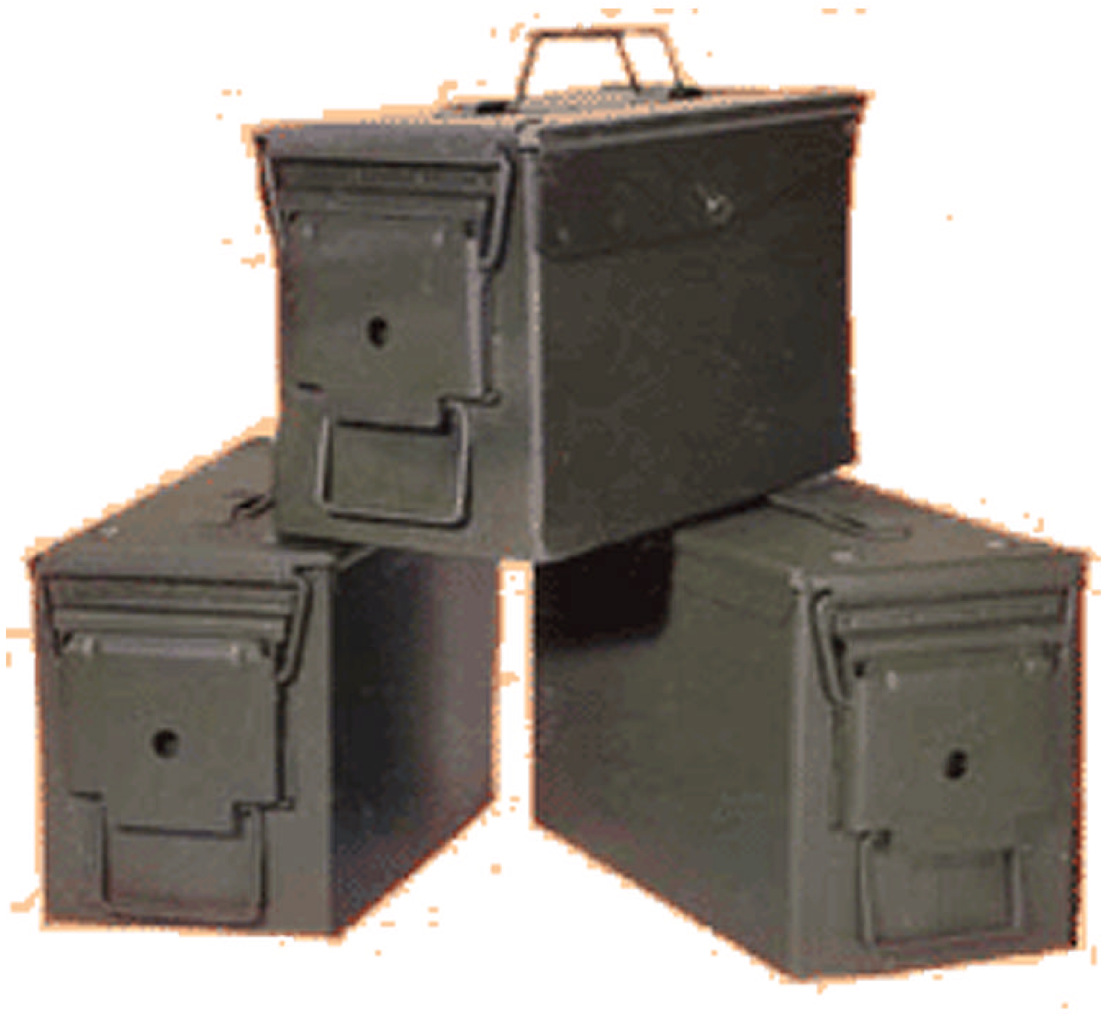


Figure 3.14 The small image of the ammo box was enough for me to build a high resolution texture from the ground up.

1. Start with a new Photoshop file; make it 400 x 400 pixels, RGB.
2. Fill the background layer with a green (I used RGB 96, 107, 86), and divide it in half horizontally and vertically using the guides.
3. Create a new layer and name it ‘Handle’. Apply the layer effect Emboss and change the Depth to 2 and the Blur to 4.

   Now draw out the handle shape from the front of the ammo box. A trick I used to make the handle symmetrical was to draw out half the handle and then copy the layer and flip it Horizontally. You can also draw a selection, fill it with color and then draw a smaller selection and delete the inside of the rectangular handle. If you use the guides, you can line up the selections to make the cutout perfect.
4. Create a new layer with the same layer effects and name it ‘Thing 1’. Use the same

technique to draw the things on the top of the ammo box. I deviated from the exact ammo box design and created three layers like this (I actually copied Thing 1 twice and resized it).

4. Finally there was a latch type object that covers the top part of the handle and the bottom of the lowest Thing layer. I made this with a filled selection (same layer effects as all the other layers) and used selections to chop out square and round parts of the object until I had the latch object. You should have an image that looks like figure 3.15.

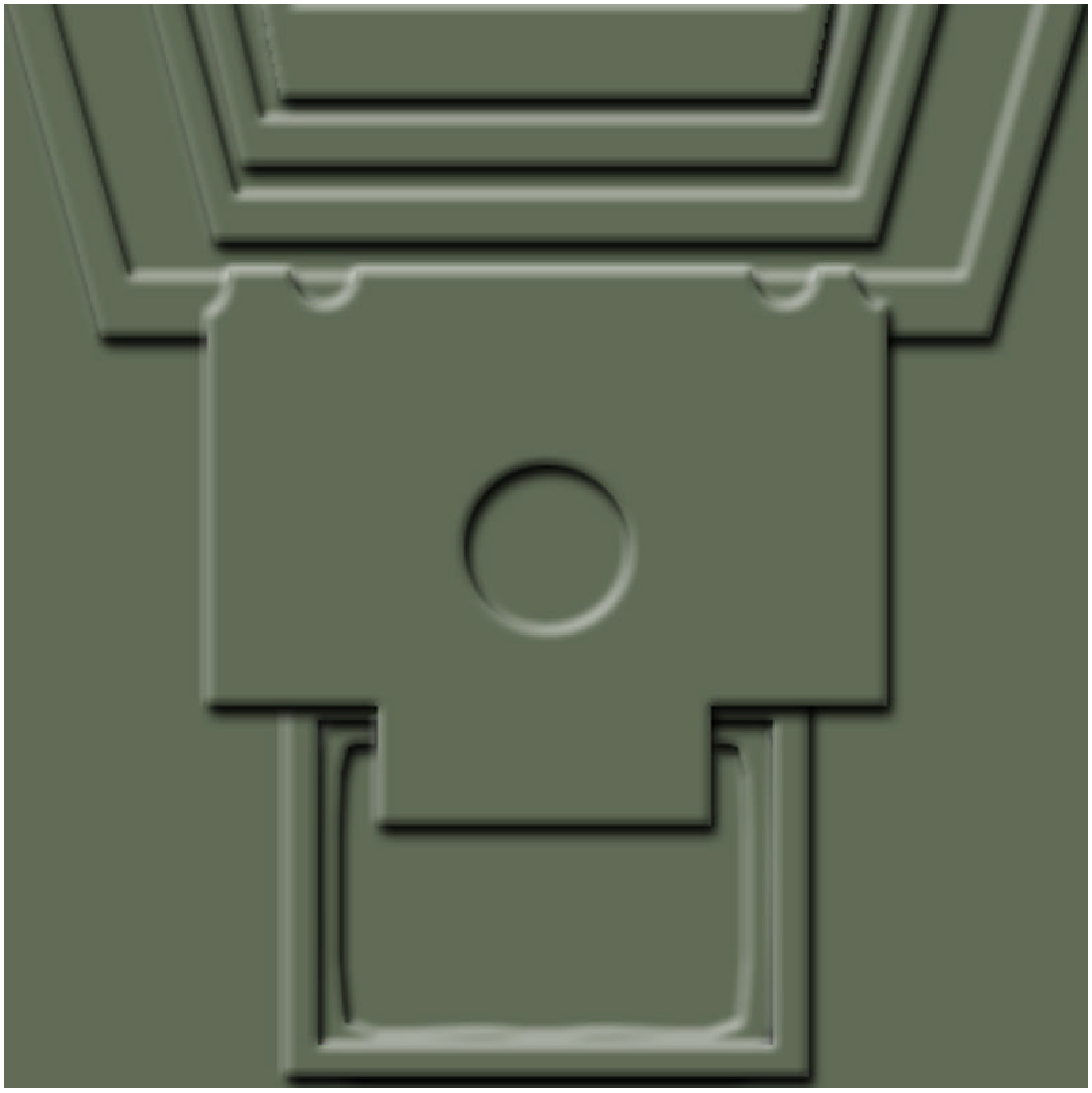


Figure 3.15 The front of the ammo box before the war, clean and freshly painted.

Adding Dirt and Scratches

The ammo box is too clean, so now we will dirty it up. These techniques we are applying can be broken down into three types of dirt: atmospheric, usage wear and tear, random stuff.

* **Atmospheric dirt** is subtle and evenly collected dirt and dust on surfaces.
* **Wear and tear** from normal to heavy usage consists of darker stains where dirty hands have been and other such contact.
* **Random stuff** is the stuff that always happens but with no real pattern involved; for example, the object was dropped, splattered by mud, or left in the rain.

To make this ammo box really look worn out, do the following:

1. Make a copy of the ammo box file you just created, and merge all the layers except the background.
2. You should now have two layers, background and the merged layer (I named the merged layer Stuff). Apply the Noise filter to each of these layers (Gaussian, Monochromatic, and Amount 5) and this will be your atmospheric layer of collected dirt.
3. On the background layer, airbrush a dark circle behind the hold in the latch of the ammo box.
4. On the ‘Stuff’ layer, apply the Outer Glow Layer Effect (Blur 20 and Intensity 9) to add more depth and help the dirtying process.
5. Create a new layer on top of these and name it ‘Worn Edges’. Set its mode to color burn and take the opacity down to 30%. Now use the airbrush, and airbrush some dirt all around the edges of the box and the latch. Use a smaller brush size to get around the smaller parts if you like. This is the wear and tear staining.
6. Finally for our random stuff dirt, I created two new layers, one named Rust and one named Dirt. I set each layer in the Color Burn mode and set their opacity to 15%. In each layer I simply pasted images of spattered food and dirt. The final ammo box image can be seen in figure 3.16.

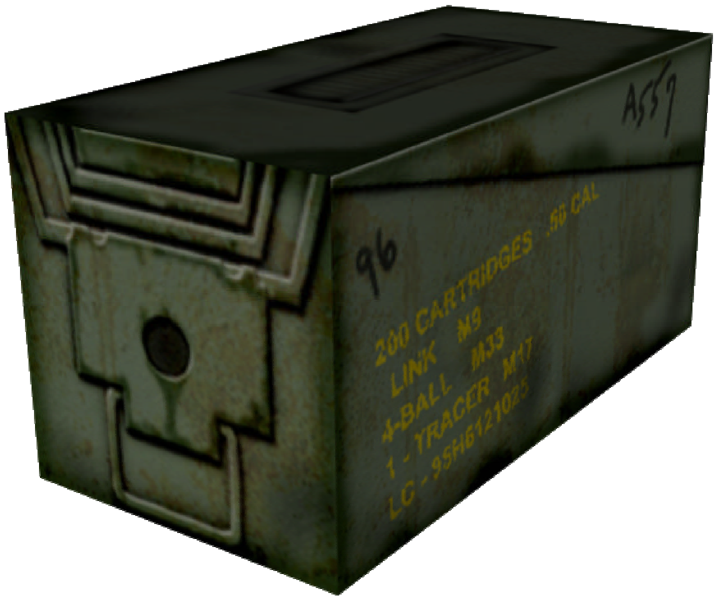


Figure 3.16 The final ammo box image on a 3D model. The top and sides were created in the exact same way as the front. For the side I simply used the mouse to do some hand lettering and the text tool to put some official looking text in too.

Using a Digital Image to Make a Sign: Removing a Flash Burn

Using digital imagery for textures is great, but the hardest thing can be to remove the flash burns you often get when the flash goes off as you take the picture. Figure 3.17 shows a sign that's 1,500 miles away from where I work, so I couldn't go and retake the photo; I had to remove the flash burn by hand and rebuild the sign. This is not as hard as it sounds. Here's how you can handle this situation:

1. I started with this image. Notice the flash of white in the middle of the sign. Also notice that the sign was photographed crooked (see Figure 3.18).

2. First I cropped the image to the edges of the sign. You can see how crooked the borders of the sign are.
3. We will be rebuilding the sign, so let’s first make the background. Start by duplicating the layer and using your Color Picker to select a color off the sign that is about the mid range of the colors on the sign. Now fill the background with this color. Add noise to this layer (Filter | Noise | Add Noise; Gaussian, Monochromatic, 17).

   This background looks plain and even, but that is how the sign looked when it was first made, even before the letters were applied. We are building this sign the way it was actually built: step-by-step – even the weathering will go on last.
4. Now create a new layer named ‘White Lines’ and apply the Bevel and Emboss layer effects to it (Inner Bevel; Depth 3 and Blur 2).
5. Make a selection approximating the outer borders of the white lines in the original sign and fill this selection with white. Apply the same amount of noise as you did to the background (Control + F will apply the last used filter). Make another selection to cut out the inside of the white area and use the guides to line it up perfectly in the center.
6. Now we’ll redo the letters, and fortunately the U.S. Army tends to use the Arial Black font on its signs a lot. Simply create three text layers in white, one for the each line of words in the sign. You can use the Free Transform option on text to get the letters sized perfectly to the sign.
7. Render the text layers and apply the noise filter to them as well. You can also copy the effects from the ‘White Lines’ layer and apply that to the text layers as well.
8. And for the grand finale, add the image ‘splat’ on the top layer and set the mode to Color Burn and the opacity to 30%.

What we just did was to start from scratch and build the sign as it was built in real life; we used a base material, painted on the letters, and then weathered it. Rebuilding the sign from the ground up was actually quicker than trying to remove the flash burn with the clone tool, or other such tedium. Not only was this quicker but the results are much better, especially for use in games. If you look at the original cropped image in Photoshop, you will see how the camera lens distorted all the letters and white lines. Trying to straighten those would be impossible. Our image has perfectly lined up lines and letters.

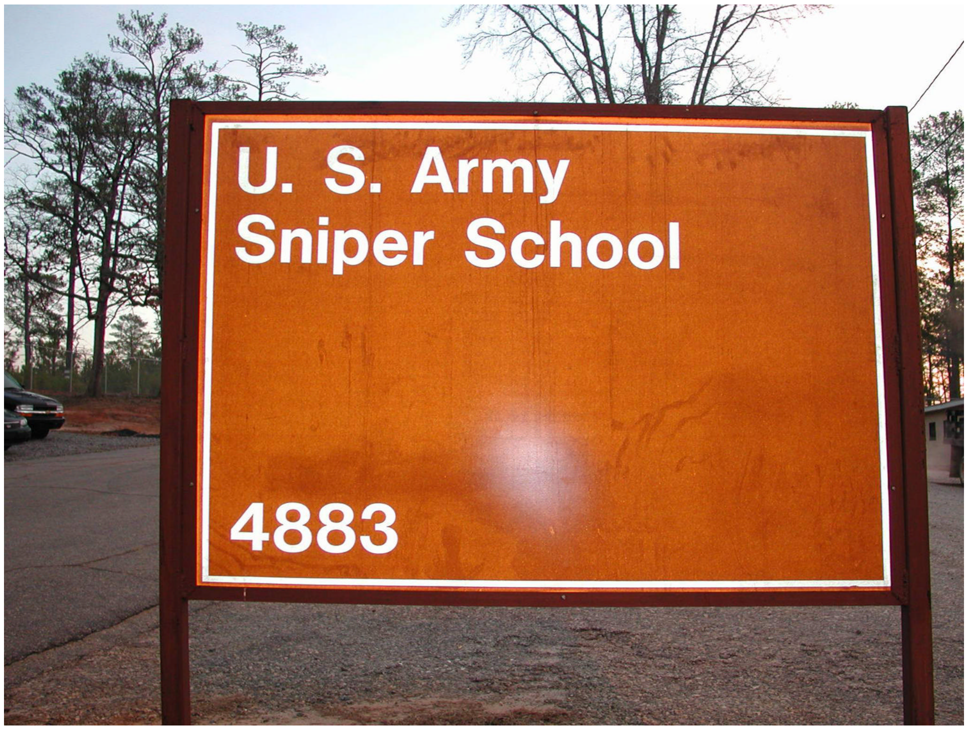


Figure 3.17 The original digital image with the flash burn in the center.

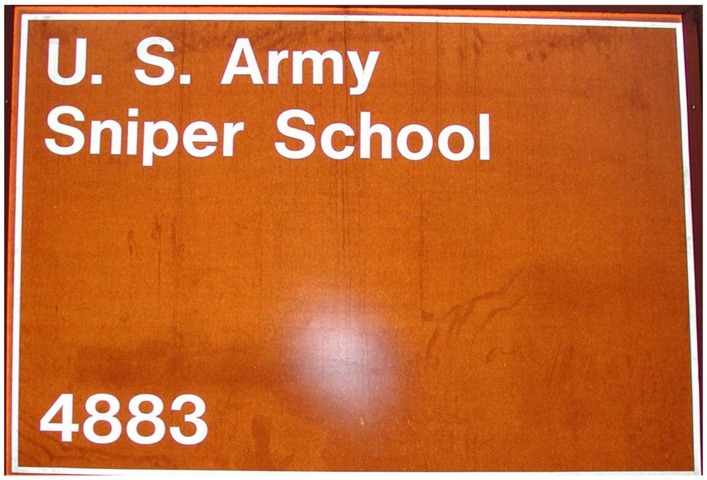


Figure 3.18 The sign cropped, and you can see how crookedly it was photographed.

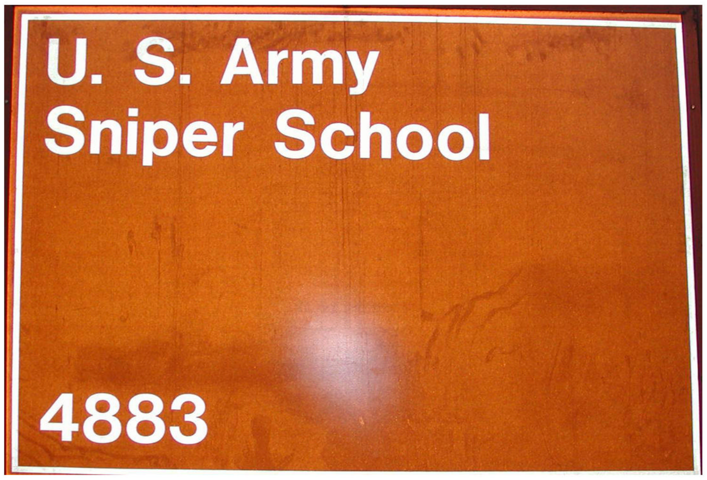


Figure 3.19 The rebuilt sign.

Creating Peeling Paint

This quick and easy effect is not only a great effect; it can make several good textures out of one base. Here's how to create the effect of peeling paint:

1. Open the image 'planks' and create a new layer called Paint.
2. We will use the Magic Wand tool on the planks layer to select an area where the peeling paint will go, but you will have to play with the Tolerance Level of the Magic Wand (I used 8 – 10) and with what color you select. This effect looks best if you don't have more than a third of the image selected.
3. Once you have the image selection in place, switch to the Paint layer and select a large soft brush and the color of paint you want - I used white, lighter colors tend to work best and look more realistic for this effect. Now airbrush most of your selection.
4. Already it looks great, like peeling worn off paint, but the final touch is to simply add the Craquelure Filter (Filter | Texture | Craquelure). You can play with the settings and get effects ranging from rough paint to paint coming off in chips. Figure 3.20 show the original plank image and the painted version.

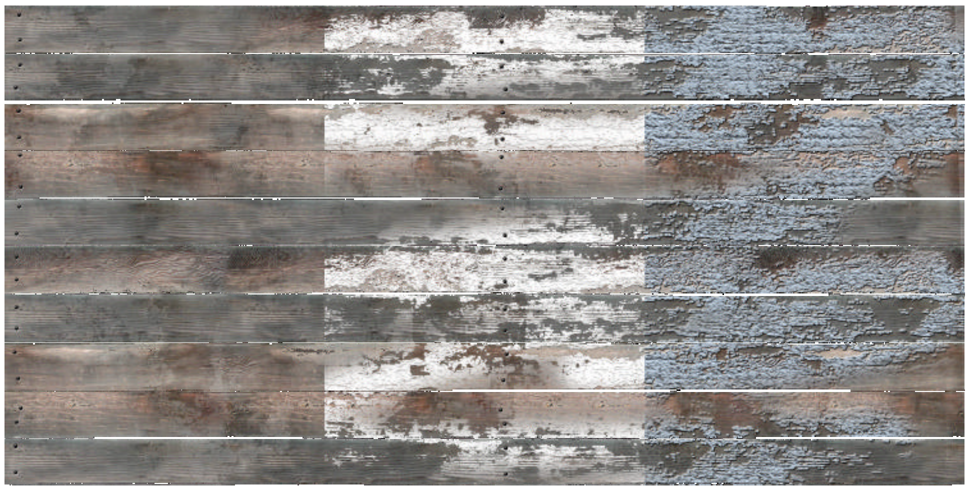


Figure 3.20 This is the original plank image with no paint (left side) one version of the paint)middle) and the more rough version of paint (right) .

Project--A Deluxe Castle Door

Now you are ready to tackle a really complex texture, but even this really cool castle door is no more complex to create than the textures you made in the exercises above. Once again, look at this image and you will see that the door is made of bases: a rough-hewn wood, a metal, and a glowing rune (or symbol). These are all simple base textures, and we will again build this door the way we built the sign, as it would be in real life. We start with the fresh wood and add the metal hinges and then the aging and weathering effects. Most objects' textures can be built using this method: create base materials, construct the object, and add effects.

Let's make the door by taking these steps:

1. Open the image 'woodplank.jpg' and notice that this is a tiled wood from the last chapter that I simply fit to my door dimensions and cropped. I even created a black background so I could add line variation by erasing some of the seams between planks.
2. Next, apply the bevel and emboss effects to the planks. The erased seams also serve to make the bevel and emboss filter effect the planks.
3. Next copy one of the planks, paste it into its own layer, and copy the layer effects from the planks as well. Rotate this plank 90 degrees and copy the layer to make the second (lower) door timber. Leave the plank a little bit too long so you can move it down and use the other side of it for the lower timber; this way the timbers won't look the same. Then cut off the excess of both timbers. Your image should look like figure 3.21.

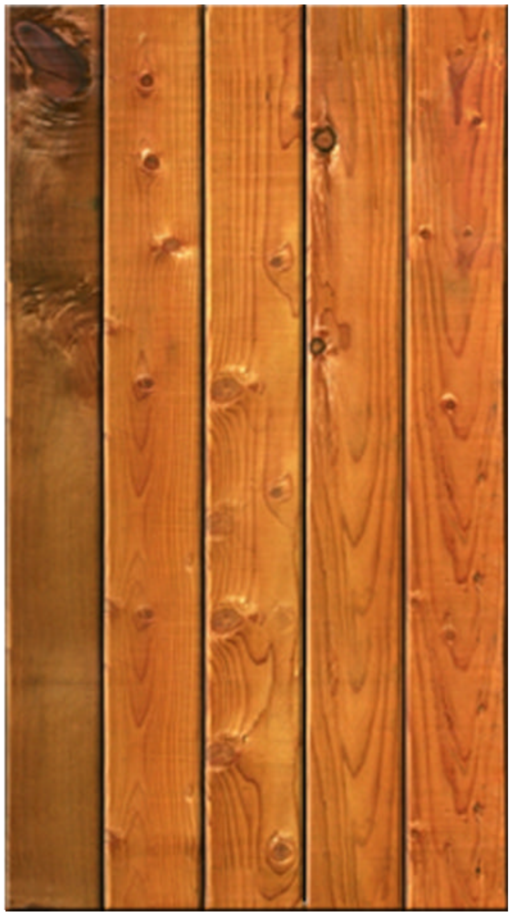


Figure 3.21 The base wood plank used to make the base of the door.

4. Now that we have the base of the door, we need to add the hinges. That fancy hinge is nothing more than the Wingdings 2 font, the lower case ‘a’. I applied the brushed metal effects from Chapter 2 on it and that was that.

   If you like to corrode the hinges a bit, run the Filter | Brush Strokes | Splatter on them and experiment with the foreground and background colors. I also added a small amount of a black outer glow to make the hinges look surrounded by some dirt as if they had been there a while.

5. Create a new layer named ‘Hinge Bolts’ and copy and paste the layer effects from the hinges to the bolts. If you are using the metal effects from Chapter 2 you should also have the Bevel and Emboss settings right for this layer. Make some small bolt heads for the hinges and cross beams of the door (figure 3.22).

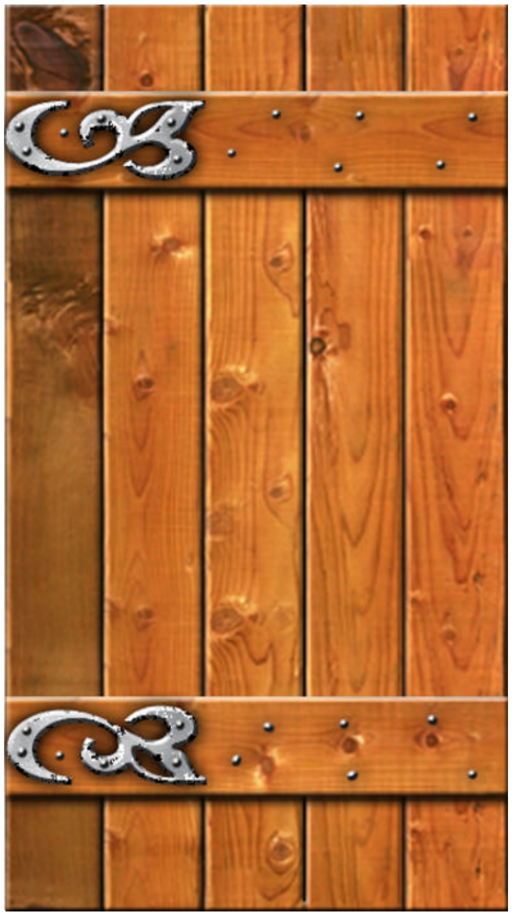


Figure 3.22 The door is shaping up with hinges and bolts.

6. To make the brass pull ring for the doorknob, create a new layer called ‘Pull Ring’ and line up two guidelines where you want the center of the ring to be. Make a circular selection, line it up to snap in the center of the two guides, and fill it with a darkish yellow. Make a smaller circular selection and center it as well and use it to cut out the center of the pull ring. Apply the noise filter to the layer (Noise | Add Noise; Gaussian, Monochromatic, and play with the amount) and then apply the Brushstrokes | Spatter filter to the layer. Paste the Layer effects in and you have your pull ring. Follow the same process on a new layer to create the little pull ring hinge.
7. Now we start the weathering. As with the ammo box, you simply create a layer called ‘Weathering’ and set the mode to Color Burn and opacity to 25%. Paint in black around the bottom of the door and the pull ring.
8. To add some mold, create a new layer named ‘Mold’ and set the mode to Hue and the opacity to 32%. Select a medium green and set your airbrush to Dissolve. Paint some mold down your door.

9. For random stains and drips, add a layer called ‘Stains’ and set the mode to color burn and the opacity to 40%. You can use black and red here to make some splats and streaks and drips from the top of the door to the bottom. Use the Smudge tool to make some of the stains look smeared.
10. To add that “barbarian wants in your castle” look, add some gouges. Create a new layer named ‘Ax Gouges’ and leave it in normal mode. Set the opacity to 44% and select your airbrush. Set the airbrush’s Fade to random steps between 12 and 55 for this exercise. Set the layer effects to Bevel and Emboss | Inner Bevel. Settings should be Depth 6, Blur 5, and the direction needs to be down. Now make some varying length and thickness marks, they will appear to be gouged into the door. Figure 3.23 shows the progress thus far with the gouges.

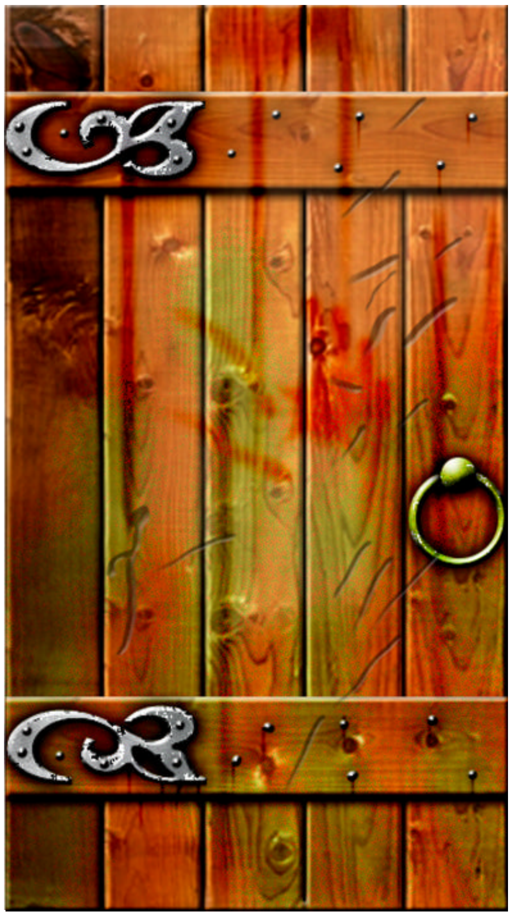


Figure 3.23 Weathered, molded and stained, this door is looking good.

11.And then we add the final layer of stains and junk. Create a new layer named ‘Stains’ and paste in the same texture from the ammo box and set the mode to Multiply and the opacity to 79%.

12. Now we are ready for the final touch, the magic rune — which is simply the upper case ‘H’ in the magical Wingdings font. Make the letter white and add the Layer Effect Outer Glow. Make the color of the glow yellow and the settings as follows; Mode Hard Light, Opacity 83%, Blur 16, Intensity 363.

And there you have your deluxe magical castle door as seen in figure 3.24.

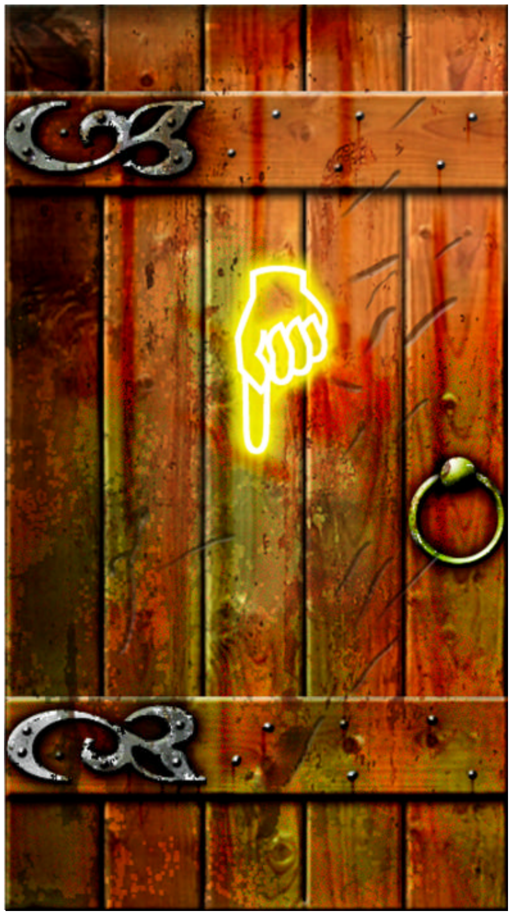


Figure 3.24 The final version of the deluxe castle door.

Moving On

In this chapter you learned how to organize textures and determine what textures you need to create, and we learned how to create those textures. Learning how to break scenes, objects and anything you see down into their base components of materials, lights and shadow, and other effects helps you create anything from the ground up.

In the coming chapter we will look at the logo, possibly the most important piece of art that represents your game, company, or project.

Chapter 7 World Building

There is a difference between level design and level construction. *Level design* requires an in-depth knowledge of the specific game, game world, genre, and audience. This chapter focuses on *level construction*—building a level that looks good and runs efficiently.

Understanding Level Construction

There is a difference between level design and level construction. *Level design* covers actual game-play issues that a designer need be concerned about, such as mission design, story line, game balance, AI (artificial intelligence) paths, and other aspects of designing a level as a world for the specific game to be played in. Level design requires an in-depth knowledge of the specific game and its game world, genre, target audience, and other related facts on an per-project basis. In this chapter, we will focus on *level construction* — building a level that looks good and runs efficiently.

Balancing Art and Technology

Looks good and runs well—that sums up the balancing act that you'll be involved in when you build game levels. Level construction involves the constant trade-off of art and technology. Typically, when you're designing a level, you will begin by gathering your ideas--first sketching the layout of the level for game play and aesthetic reasons. You will draw a top-down map of the area to be built, and you'll start detailing buildings, streets, hallways, major obstacles, and objects. (In this chapter, we will start by sketching the levels we have planned for this part of the book, which will be simple for our purposes.) This is the stage where you can let your imagination go wild. Brainstorm freely now; later you can adapt your ideas to the limits required by the game technology.

When the game level's plan is fairly well developed, and you start translating your "analog" sketches and ideas into a digitized game level, the trade-offs will begin. When planning a map, even a demo level such as ours, you will take into consideration the technology used and your budget for assets. The word budget is used here to mean the amount of game assets you can have in your level such as textures (artwork), objects (or geometry), special effects, and any other aspect of the game level that will affect the performance of the computer. (You will learn later some of the optimization tricks and techniques for larger and more complex levels.) For example, a large level may look wonderful architecturally, but the amount of artwork covering all the surfaces may have to be limited to a few low-resolution images. This may work well in a game world set in a city where it is common to have tiling images of the same material. On the other hand, an ornate throne room where you want to have paintings, elaborate floors and walls where many textures are used to cover complex geometric surfaces, you may have to make the rooms smaller and divide them with hallways so the computer only has to draw one elaborate room at a time. Compromises are a constant juggle; a few high-resolution textures or many low resolution textures, small geometrically complex room or large more plain ones? Any aspect of your game that takes system resources to handle needs to be balanced because on one hand you want as much stuff in your level as possible, but you don’t want to ruin the game play with choppy frame rates.

Knowing the Level’s Purpose

Of course, in level design, you will primarily want to know the purpose of the level to even begin planning it. Is the level a death-match level, a single-player level, a capture-the-flag level, or even a tutorial level? Many games have simple levels in the beginning to teach players the game play and functionality of the title, and these introductory levels are designed differently from the in-game levels.

For this book, a small level was designed to showcase the tools and techniques of 3D game and level construction. In order to use the cool textures we built in Part One, I built a small castle level that is technically an indoor level, but we will also see some sky in an open courtyard. Building the castle will introduce us to the basics of Genesis 3D and to some of the finer points of level construction, such as texture planning, object placement, and basic lighting. Simply getting your first level to run is quite an achievement.

Deciding on the World Setting

Location, time period, and *atmosphere* are some of the most critical elements of a level for a designer. These aspects contain the information you will need to start the artistic design of a level. You can then begin to think of details such as the architecture of the world, ornamentation, weathering, and aging, as well as other elements that will affect the textures, geometry, and even the way the sky looks. But before you start on the details, you need to know the basics:

* Will the level be an indoor or outdoor level? Most game engines can’t handle large outdoor terrain and if you plan to have any terrain you will be sacrificing detail in many areas.
* Will it be a medieval fortress or village, a modern city, or a futuristic space port? Historically (or fictionally) time period and culture can dictate the complexity of buildings and objects, the colors (and amount of color) used, and can give the designer clues on how to use that setting to their advantage. Egyptian settings can be made of large blocky walls that are covered in a base sandstone texture. Complex temples or statues can be placed far enough apart that the computer will only have to draw them separately and not all ot once.
* Will you be building a realistic environment or a fantasy creation? This is one area where what we can see can be used to our advantage. In a realistic setting like a mall we may have a clean view far into the well lit building, but in a dark dungeon of the same size we will not be able to see nearly as far. Since the

computer doesn’t draw what is in the darkness we can have pools of light with high detail and darkness in between to break up world size.

* What will the color scheme, the mood, and even the weather be like here? Every detail of the world will can be both a challenge or used as a trick to help the designers. With color scheme that is muted and low color we can compress textures more, with fast paced game we are showing the player more of the world faster and the computer gas to keep up, but if the game you are designing is a slower paced walking game than you can afford to introduce more to the player in the level and the computer will have an easier time keeping up with the demand. And weather, which can be hard to create in a computer, can also be used to help engine speed. Like darkness, fog can be used to set mood as well as obscure what the player can see so the computer only has to draw what is closer to the player rather than what is all around him in the entire world.

Adapting Your Art to Fit the Game World’s Perspective

As you start building the background and objects in a level, you will soon discover that some things you thought would be simple turn out to be a little more complicated than you anticipated. For example, even the size of a door, which may seem like a simple detail to obtain in the real world, is a challenge that requires much experimentation and input to determine. For example, in the game *Unreal* (created by Epic Games, Inc.; see **www.epicgames.com**), the characters are cartoonish, the doors are large for quick fighting in the first-person shooter (FPS) style, and the architecture of much of *Unreal* is very sci-fi. If you’re trying to build a modern building as a setting for a game using the *Unreal* game engine, you will be faced with characters that are too large and that may not fit into the doors you build. And the perspective of a door that is built to real-world proportions will not look right as viewed from the eyes of a game character in *Unreal*. The view from the eyes of most FPS characters is a bit fish-eyed to make up for the fact that the player is looking at a flat screen and has no peripheral vision (the scene ends at the monitor’s edge). Eventually you realize that the door has to be experimented on a lot.

Walls are the same. You may find a wall height that looks great in the level editor and the game, but from a texturing point of view, when trying to lay the texture on the wall (that will most likely have to be a power of two as discussed previously in the book), you find the texture does not line up right without a lot of resizing by the level editors. So back to the drawing board you go to try and make a power-of-two texture that fits nicely on a door when applied, the balancing act of art with technical limitations goes on.

Planning the Level

After you know the level’s purpose and general environment (location, time period, atmosphere), you can start designing and planning. For our castle project, we will start by making a few rough sketches of the castle, the floor plan, and a couple of detail views just to be sure that our ideas work on paper before we dedicate a lot of time to an idea that may not work. Sketching, in even the roughest and most preliminary way, also helps to keep you from drifting in your design. Although you can see that the sketches made for this small project are quick and not very detailed (see Figure 7.1), they helped nonetheless in laying out a set of rooms.

This phase is important. When working on screen in the level editor, most people are operating in a different mental arena — needing to be so much more patient and precise than they need to be on paper. Your best ideas for the level may not even surface if you don't get them on paper. Even the roughest of sketches will improve what you build in the level editor because, when you're sketching on paper, you can explore ideas unfettered by the tedium and the limits of the level editor. Experimenting while sketching an idea on paper is easier because we are not investing as much time and effort as we would if we were building it directly in the editor. Every minute you spend in planning, sketching, and giving thought to your level or room is reflected in how much better your level will look in the game. As you will see in the coming chapters as you go through the process of building a level from the ground up, the difference between a good level and an amateur-looking level in actual level editing time is sometimes only a matter of a few minutes.

In addition to giving you more freedom to explore your ideas, sketching them on paper before going digital can make your final level easier to build. When you are working from a well-thought-out sketch, you can sometimes put together a level extremely fast. Because you are not thinking of so many things at once, you can think in a one-track fashion; the ideas have already been detailed, so you are focused only on implementing them.

The Castle Plan

This is the floor plan of the castle level we will be creating for this book. A few rough sketches convey the layout, and as you can see in Figure 7.1, the floor plan is rather simple. But even this simple layout benefits from planning. Once this level is completed--when it's textured, when architectural details have been added, and when lighting is completed--it will look great.

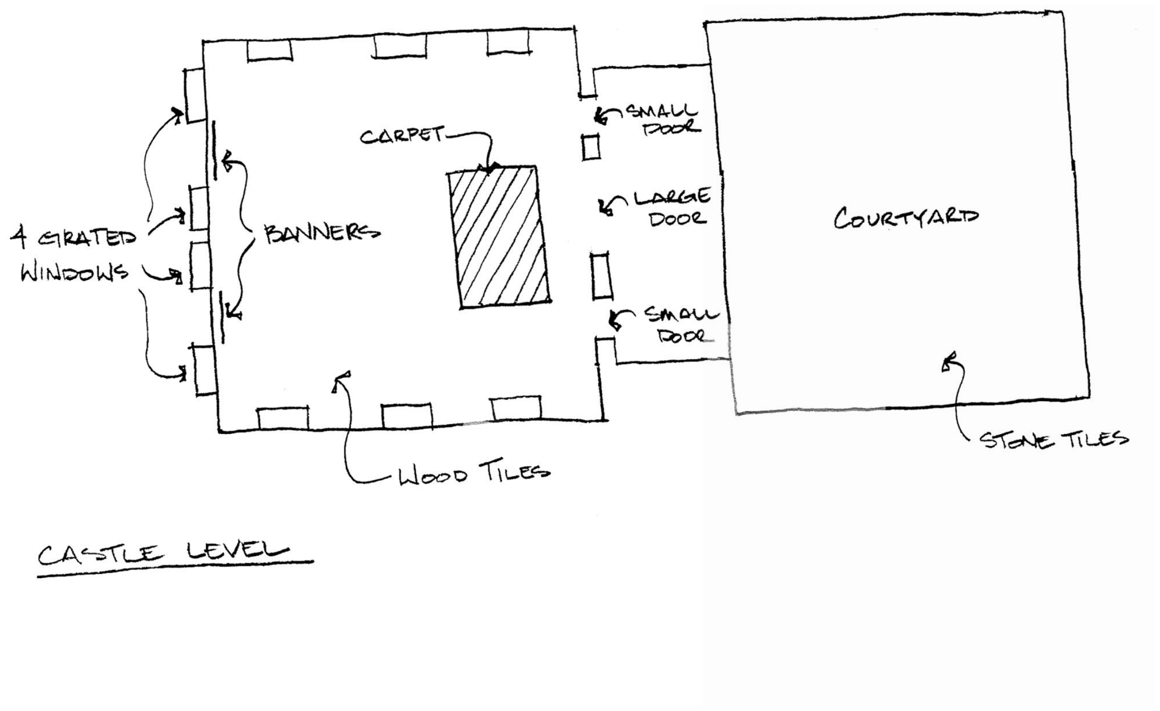


Figure 7.1 This is the floor plan of the castle level. Notice how simple the rough sketched plans are.

Understanding the Level Development Cycle

Typically, a level is developed in the following fashion: Keeping in mind that developing a level is only part of developing a game. Level design and development has a language all its own (as does most arts of game development) and in fact most sizable development teams keep a glossary of terms as part of the design document. Since game development teams are frequently made up of individuals from schools, other professions, small and large development teams that have all developed many diverse types of games we all have our own word for different concepts and processes. The steps below are as typical as I can get, but are still according to my experience.

1. With basic information known about the level from the design document (its purpose, its general environment, the weather, and the story surrounding the level), designers can start sketching and storyboarding. Ideas are refined, and decisions are documented. Procedures and processes, documents and worksheets are created that will make the level creation process smoother and faster alter on in development, as well as make the levels play more consistently in the game.
2. The level is mocked up (on the computer) from the documentation. The initial prototype is produced, people applaud, and then everyone starts suggesting changes. The processes are refined and should be solid by this point.
3. The base texture set is created or added to the level as well as other assets such as base sounds, animations and models.

4. The level is tested, fixed where necessary, and approved (repeat this process until moving on). At this stage it is also common for levels to be reworked a great deal, redone totally, and sometimes even scrapped altogether. It is also common that a poorly designed project hits major trouble at this stage of implementation when it is discovered that nothing works as planned and the whole project has to be reevaluated. Sadly this happens often to small developers and they usually don't have the time and money to start over or spend weeks redesigning the game.
5. The level is finalized. Lighting, entities, optimizations, and continuity checks are complete. Not much applauding happening at this stage as people are working hard. Suggestions are hopefully at a minimum by now too.
6. The final level is compiled, tested, recompiled, and released.

Note: Remember that level construction is only part of the overall level development. And even before a formal testing stage occurs, the level designers, programmers, and artists may all have a level they call a test level, used for benchmarking and testing various aspects of the tools and technologies they are using.

Ideally, before a level is constructed, a great deal of the game is already understood on paper. Sometimes, however (many times, actually), artists, modelers, and level builders are put to work building levels as the design document is still being written. This is usually how it happens, and often it is because the development company has the technology ready to go and wants to keep the team busy. Sometimes, however, it is due to the fact that the game is not being properly developed. The best way to construct a level is when you are provided with a folder that details the storyline, setting, mission goals, art budgets, timelines, maps, sketches, and a wealth of information that is essential to good level construction. Often, though, a level builder's work consists of redoing work, adding elements after the fact, and even redoing whole levels (if they are not dropped from the game), and these last-minute additions and changes should be avoided if possible.

Prototype Level

Usually a prototype level is built from an idea that has been laid out on paper based on the documentation previously generated. This layout may be solely the creation of the level designer, based on notes and the design documentation, or the level designer may be given sketches, floor plans, and a mountain of information both textual and visual that must be assembled into a game world level. This process may require several stages of development depending on the size of the level, the size of the development company, and the degree of structure the company uses. Some development teams have one person doing all the "level" work, and other teams have level designers, level editors, and even specialists in lighting and game play. Regardless of how level design and construction are done, it is an evolutionary process either up front or during the actual construction.

Usually prototype levels use assets that are placeholders, sometimes levels are without textures and are nothing but box mock-ups of the playing area. But even if a level is complete, it should be constantly reviewed during development. You do want to avoid redoing work at all costs, but as you use the tools for a year, gel with the team, understand the game, and improve your skills, the level you built at the beginning of development will most likely pale next to the one you built at the middle or end of development.

Actually, even though “levels built” is a milestone or goal in most development schedules, and the process looks linear on paper, in actual practice it shouldn’t be. In order to keep the team and the levels fresh, it would be wise to have a good bit of *coordinated* level hopping. If one level presented a challenge that the team worked on for weeks and the challenge was met, but the level is not complete, it may be wise to let the team members get a breather and switch gears to a more interesting level. This also ensures that all levels are kept up to date. But remember, this process must be organized and controlled, or there will be chaos as different team members go in separate directions.

Test Level

After most of the paper design has been digitized, tested, and discussed, bugs have been fixed, and other major obstacles have been addressed that may cause redevelopment later on, a test level is built that closely emulates the final game. After the paper has been updated, the levels are laid out in detail and construction begins. At this point, there should be no more questions about the game and its design, atmosphere, setting, or mood. What’s left is producing the game and polishing it, making it work as defined, and adding no more to the mix if possible.

At this point the process for building and testing levels should be in stone. How assets are created, named and stored should be standardized and documented. But it is almost impossible for even the most experienced level designer to think everything through up front. So very often, even after the game is running and playable, mistakes are found, dead spots in levels are discovered, and other areas that must be addressed become known and the changes must be reflected across the board in development.

Level development should always be subject to a process so that the level is consistent, the level is running at its best, and the workflow is paced to keep the team from being bored or overworked.

Consistency within a level is more than making sure that textures and other elements are similar from one scene to the next--that castle floors don’t change from aged stone to new brick, for instance. Consistency can also involve items from the game’s story. For example, say that the story says, “Legend has it that the castle with the green stone walls contains powerful magic.” Someone has to make sure that the walls of the magic castle are in fact green.

The Demo Level

The demo level is the level that testing firms, reviewers, and sometimes the general public get to see. It should have no obvious or critical bugs or problem areas, and it should represent the game well.

Mistakes to Avoid in Mapping Your Levels

To begin with there is a great article on level design "The Art and Science of Level Design," by Cliff Bleszinski of Epic Games at **www.cliffyb.com/art-sci-ld.html** that you must read. In this article Cliff Bleszinski says the following, "In addition to having dedicated world texture artists and environment concept designers the need will soon emerge for dedicated "prop" people; artists who create content that will fill up previously static and barren environments. Most architecture is relatively simple, much of the detail in the real world comes from the "clutter," the chairs, tables, and decorations that fill these places up." And we are already there. Currently there is a scramble to hire more specialists who can model world objects, certain types of world objects, and texture them.

He also states, "It is very likely that the level designer will be like a chef, taking various "ingredients" from other talented people and mixing them into something special while following the "recipe" of a design document. Right now there are companies that have artists lighting levels, as well as doing custom texture work on a per-surface basis. The level designer will evolve to the role of the glue of a project, the hub at which everything comes together." And this is exactly how the current project I am working on is setup as well as many others development efforts starting up.

In this same article he also lists his 'Design Commandments,' which I summarize below.

Designer, Evaluate Thyself

The best level designers are never afraid to step back and re-evaluate their content and take a break from it as well. Once you become burnt out or sick of your own level you are in danger of doing bad work.

Thou Shalt Seek Peer Criticism

A great designer is never afraid to take criticism from his peers; in fact, a great designer is the sum of himself plus his peers. Prima Donnas are often the weak link in a design team. The ideal designer seeks criticism even from those he may consider "less talented" than he, because even if he believes that the critic in question has no skills the commentary will be fresh and from a new perspective.

Thou Shalt Value Rivalries

In addition to taking suggestions from one another it is key for level designers to feel a desire to "one up" each other. Healthy competition in any area of a development team means improved results.

Do Thy Homework

As much of this work is moving into the realm of the Art Director and art team, the designers remain the Digital Architects and they will still be responsible for much of the look and feel of the levels. Therefore, if a project calls for an accurate Roman Empire then everyone had better be doing his or her homework. Having a shared directory of R+D images on a server as well as an art bible that is referred to all designers and artists will contribute to a more consistent look and feel.

Thy Framerate Shall Not Suck

If the designers are working with a technology that can push 100 million polys then they’re going to try to make the technology look like it can push 3 times that. Although much of the framerate issue falls upon the programmers, with optimizations and level of detail technology, it is extremely important that designers have hard coded guidelines for framerates, detail levels, and RAM usage.

Thou Shalt Deceive

Pay no attention to the man behind the curtain. If a designer can simulate a newer technology with some trickery then by all means allow and encourage this. If the programmers are exclaiming things such as "I don’t remember programming that!" or "How did you do that?" then something special is going on. If a scene can look more detailed with creative texturing then go for it. If bump mapping or specular highlighting can be faked even though the engine does not "truly" support it then why not? Only the hardest of the hardcore gamer will know the difference.

The Three Sins of the Beginner

Many beginner-made levels suffer from many common ailments, things that a few minutes of tweaking or extra work could fix. The most common mistakes include bad textures or texture placement, blocky construction, and bad lighting. We will look at all these below, but if you have read part on of this book bad textures should not be a problem for you.

Bad Textures

Usually textures created by amateurs are not created properly and/or they are not placed properly on the world surfaces. Fortunately you know how to do both as we discussed these points earlier in the book. Textures are very important and are second to lighting only because with lighting you can control so much of the color and atmosphere of your level with your lighting.

Textures are abundant on the Internet, as are tutorials and tools to make textures. (Remember not to use other people’s textures without their permission.)

Taking the time to place textures properly in the game world pays off in a big way, as you will see. You may notice that a really amateur level will feature textures that are simply colorized noise, hard lines, and other techniques that are produced by people that have no texture creation training. Figure 7.2 features bad textures and bad lighting in the same scene. The other problem with amateur texturing is the lack of attention to detail. Seams that don't meet, stretched or squashed textures, and textures with no detail placed quickly look terrible.

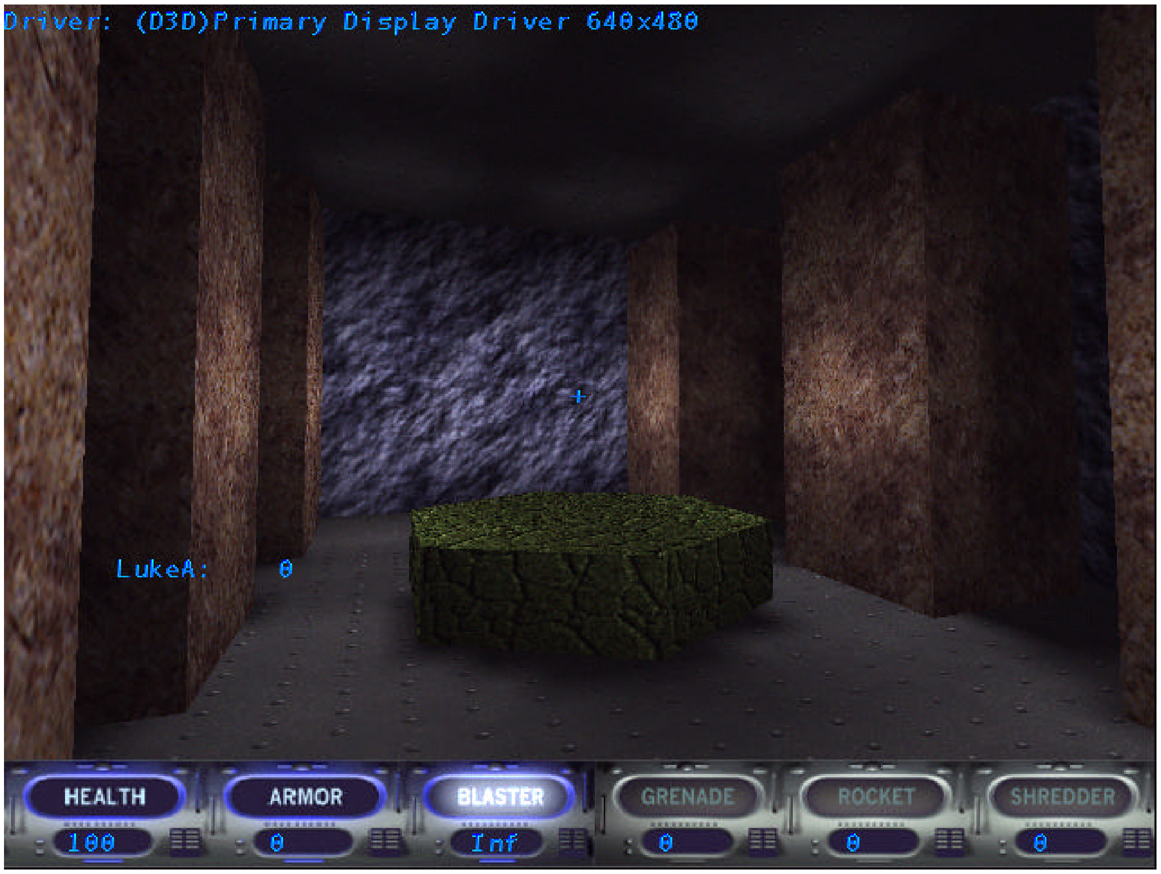


Figure 7.2 Bad textures. The textures lack detail and depth.

Blocky Construction

Levels with blocky construction feel as if the world was built rapidly with no attention to detail. Simply dropping cube about a map is not level building, making the cubes look like something in a real or imagined world is. The fact that Genesis has no vertex manipulation, or the ability to drag out individual points on an object, makes blocky construction an obstacle. You can taper and skew cubes, for example, and make a cool base to a column by typing in different parameters when creating the cube, but that is not vertex manipulation. There are ways around this limit, and using light and composition is one of them. All I did was to skew or slant the columns inward, but that adds a sense of the dramatic from simply having straight square columns. See Figure 7.3 for an example of a level with the same objects arranged and lit differently.

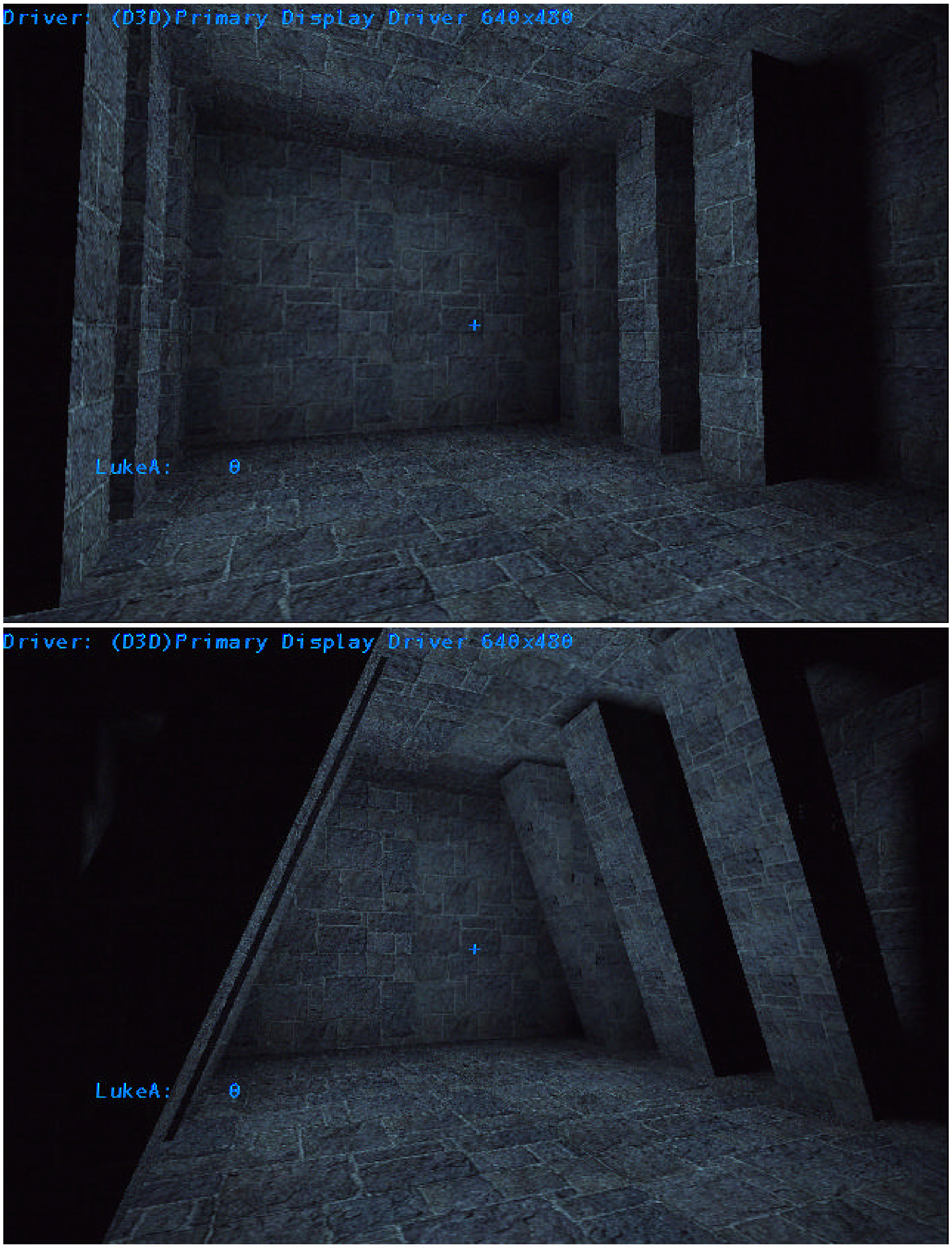

Driver: (D3D)Primary Display Driver 640x480
LukeA: 0
Driver: (D3D)Primary Display Driver 640x480
LukeA: 0

Figure 7.3 This is an example of a room with the same columns only one set is slanted to give the room a more interesting look. Composition is an important aspect in making levels look good, especially since we are working with fewer and simpler objects.

Vertex manipulation, as we said, is the ability to drag out points of a 3D object (figure 7.4) and change the shape of the object. While Genesis does not allow this, there are a few things you can do to get around this limit. One is simply typing in different values. In figure 7.5 you can see a normal proportional cube and then a cube that was created by typing in the top X and Y values to make it tapered – we will look at this more specifically in Chapter 9. Figure 7.5 also shows the same columns with texture and lighting so you cans see how much better this simply trick looks in use.

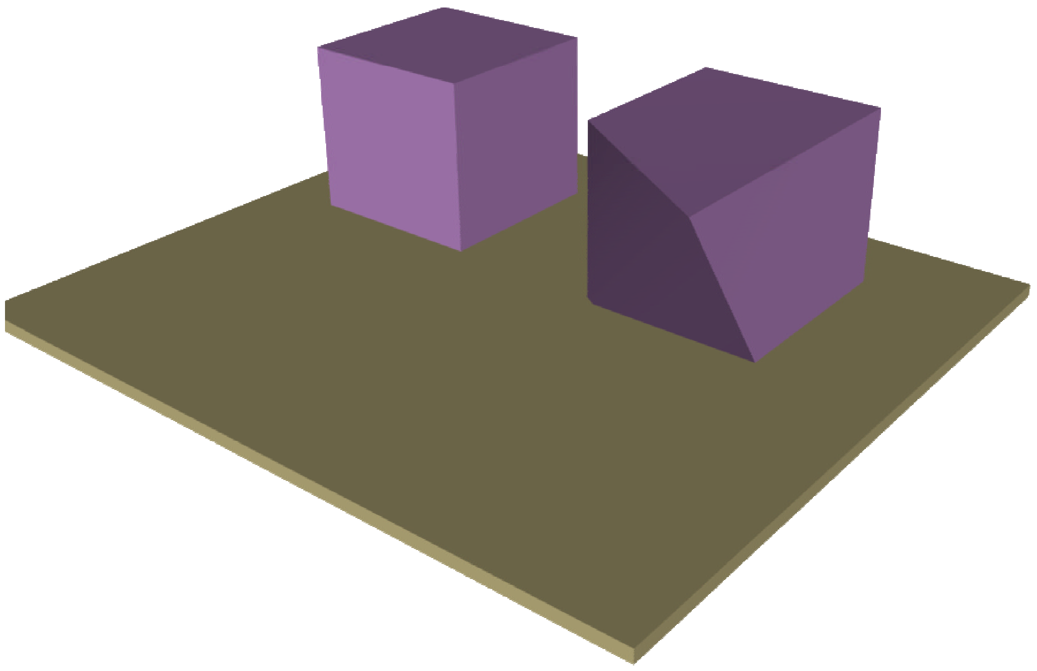


Figure 7.4 A normal cube and a cube tapered.

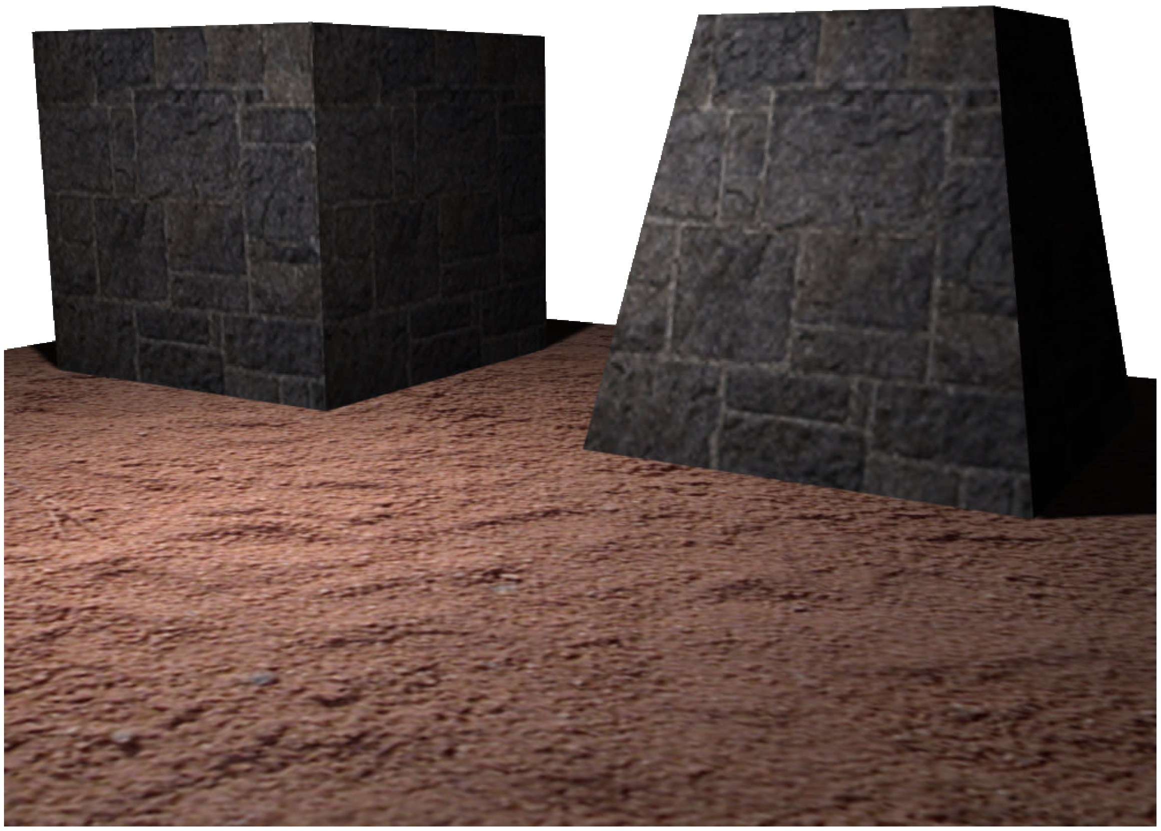
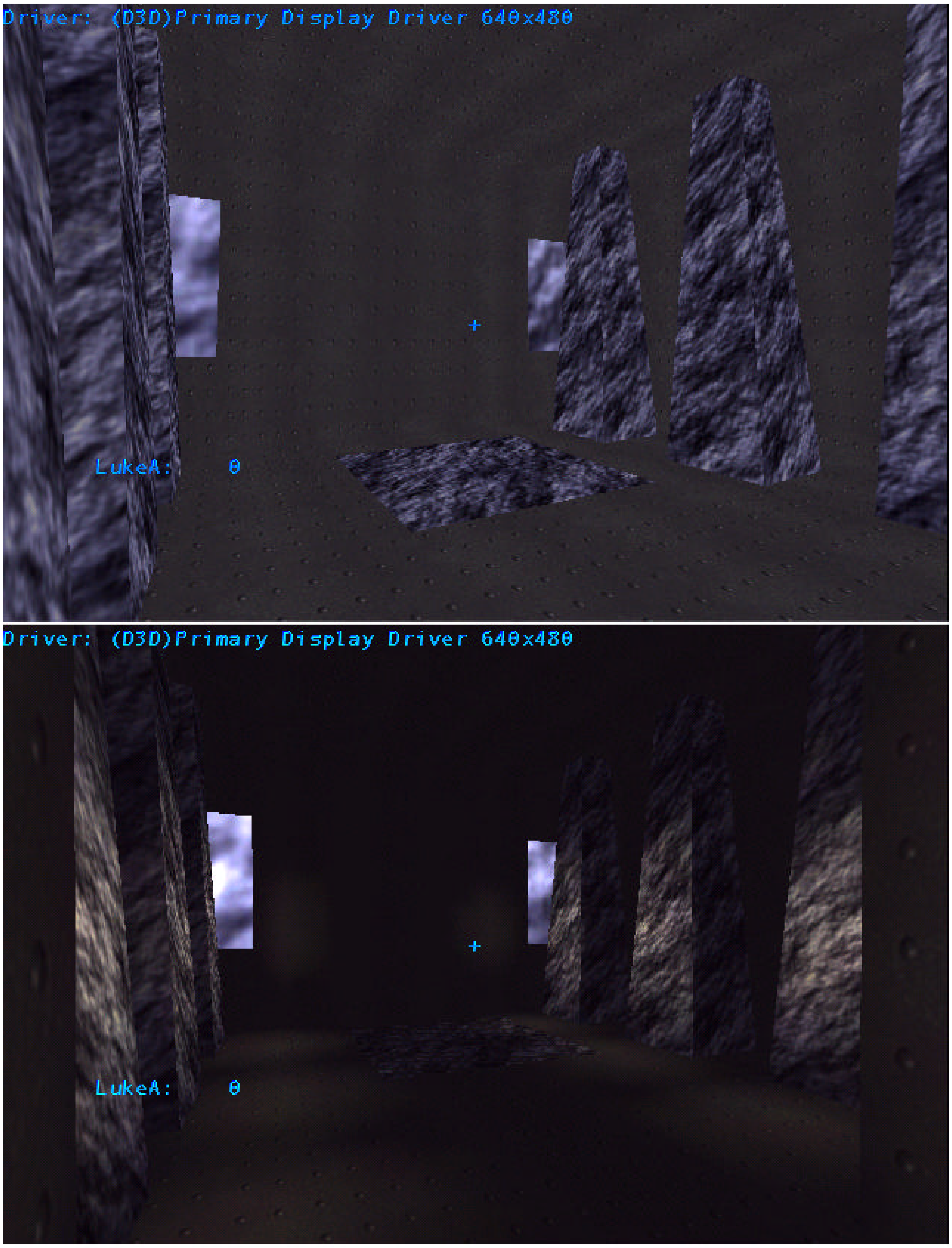


Figure 7.5 The same cube with texture and lighting.

Bad Lighting

It is possible to make a well-lit series of rooms with *no* textures look, or rather feel, better than a poorly lit room with good textures. With poor lighting, your level will look like a bunch of painted cardboard boxes, as seen in Figure 7.6.

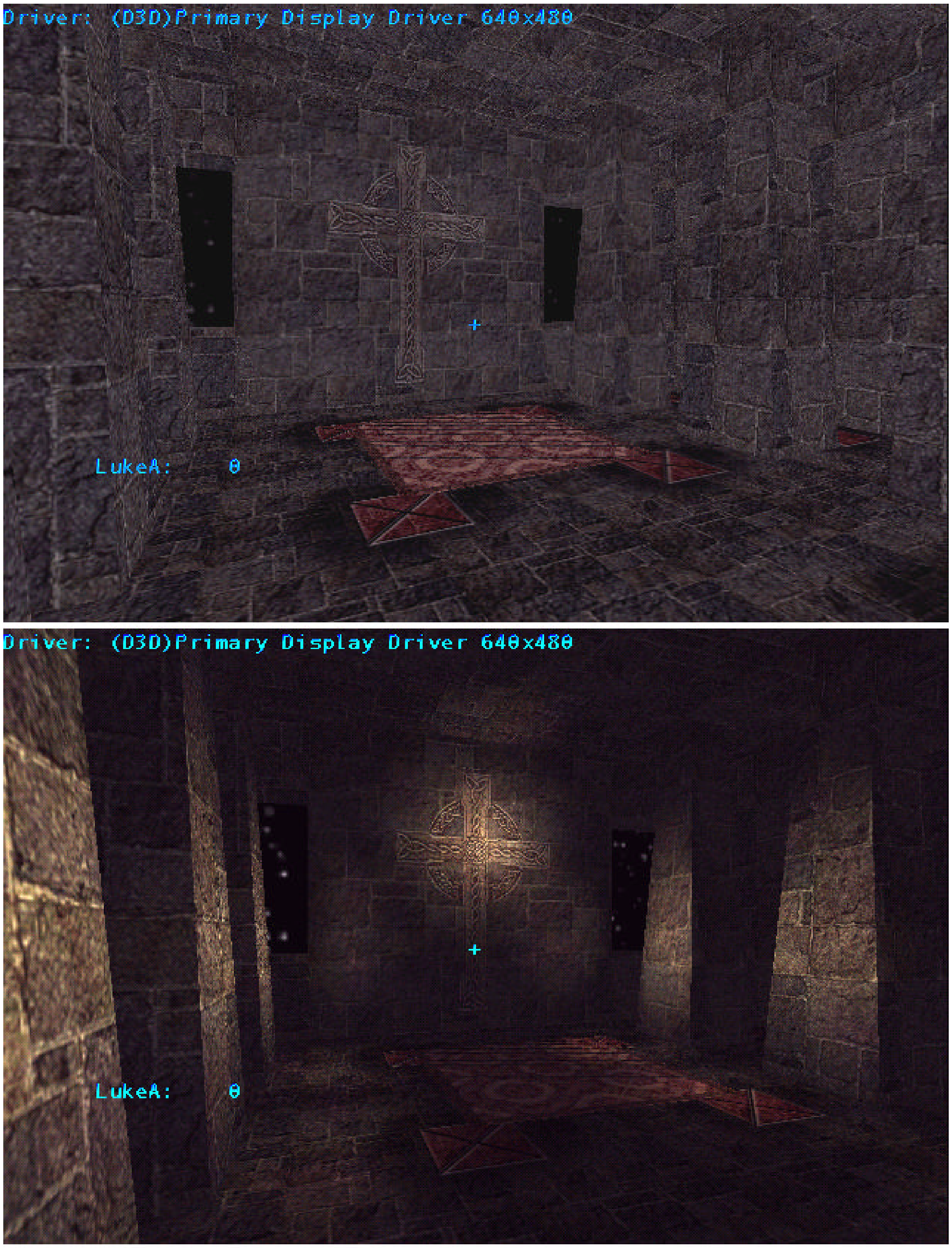


```
Driver: (D3D)Primary Display Driver 640x480
LukeA:     0
```

```
Driver: (D3D)Primary Display Driver 640x480
LukeA:     0
```

Figure 7.6 Here is a level with bad textures and poor lighting (the top image), and the same level with the same textures and good lighting (the bottom image).

You will see that the default light level in Genesis (as in many game engines) is harsh and flattens the colors and depth of the scene. The only benefit to this default setting is that level designers won’t find themselves in the dark if they forget to place lights in the world. One of the most powerful aspects of Genesis is the ability to control the lighting and its many parameters. Lighting in Genesis is a huge bag of tricks that should be explored and exploited. Unfortunately, many beginners leave the default setting of light in the world and touch no other lights. See Figure 7.7 for the difference lighting can make to a scene.



```
Driver: (D3D)Primary Display Driver 640x480
LukeA:     0
```

```
Driver: (D3D)Primary Display Driver 640x480
LukeA:     0
```

Figure 7.7 Here are two shots. In the top image, the default lighting was left on. In the bottom image, the default lighting was turned way down and a few lights were hand-placed in the scene, about 30 seconds of work.

Texture Planning for the Castle

As we have seen in Part One of this book, there are many ways to create 2D textures. They can be scanned in, come from a digital camera, rendered in a 3D program, or simply made completely in Photoshop. All the textures in the remainder of this book in all the exercises starting in Chapter 9 were created using digital images (from a digital camera) that were touched up (a lot) in Photoshop and some were made using the techniques presented in part one.

Note: All assets for the tutorials are on the CD-ROM in the “tutorials” folder.

Going back to our initial level design, the sketches, and the small discussion we had about the project previously, you will see that we have to create a few types of textures for the castle environment. We need to create the stonewalls that will cover 99% of the level as well as the floor, create wood for beams, and create detail and specialty textures such as the banners and any ornamentation in the level.

Since our castle is old and abandoned, the texture on the walls will be old stone. This texture was made from a digital image taken of stone from an old building local to me. I had to make the image tile well, which is a challenge since the stone was irregular (see chapter 2 for tiling tips). While the stone had to be cleaned up and modified in Photoshop to tile well, it also had to be made to look dirty and old. So on one hand, the stone image was cleaned up to tile well, and imperfections were removed that would stand out and make the pattern of our tiling texture visible. On the other hand, the stone image was weathered and aged. Making the texture not look tiled or too plain is not only texture work, but is also part lighting and part usage of the texture in construction. See Figure 7.8 for examples of three textures: one that is too plain, one that is obviously a tile, and one that is just right.

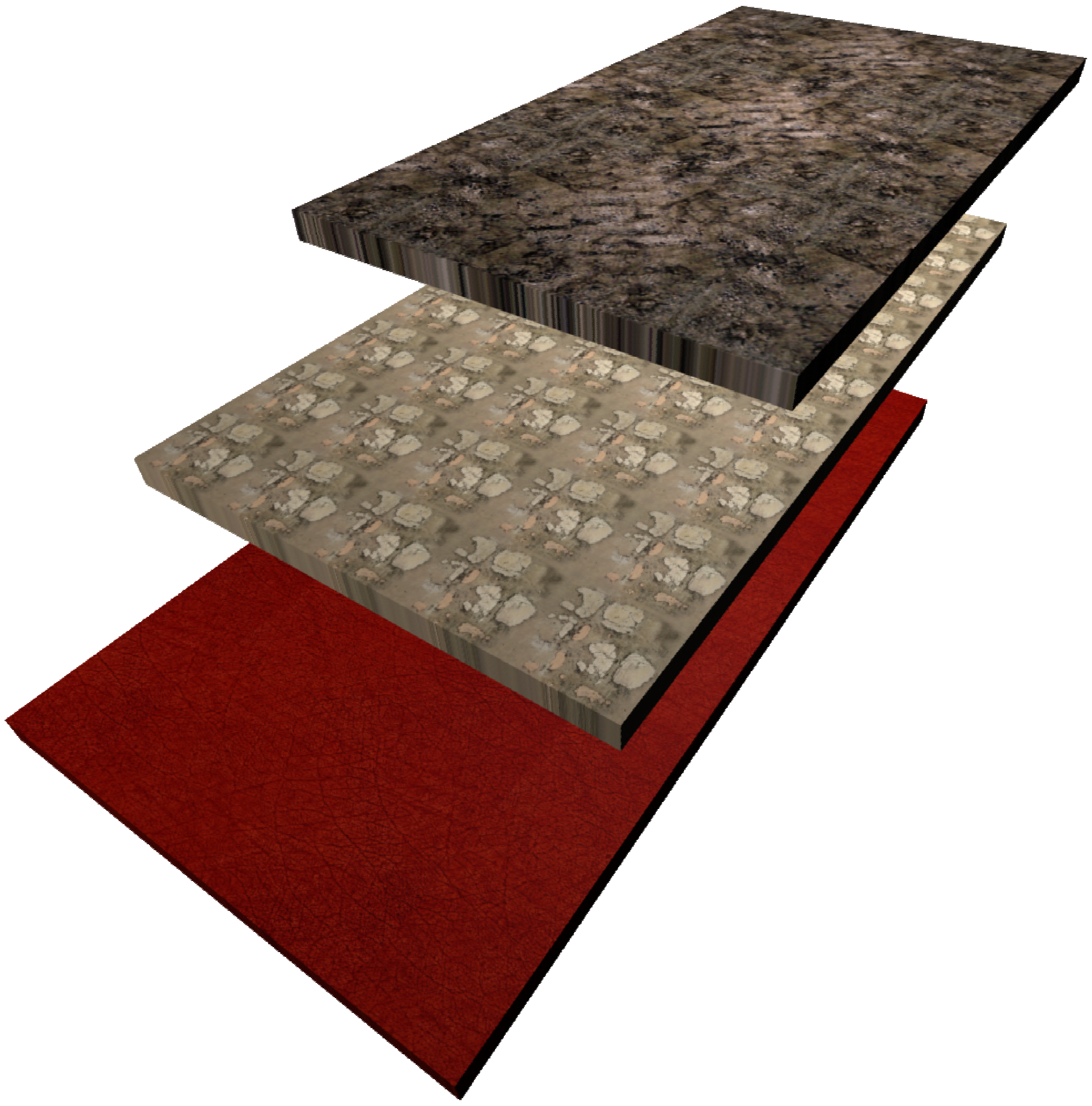


Figure 7.8 Here are examples of textures that are too plain (bottom texture), obviously tiled (middle texture), and finally, just right (top texture).

The Castle Texture Set

Called the *texture library* in this book but often also called the *texture set or* the *texture palette,* these terms refer to the set of textures in your library of textures that will be used to map the level's surfaces as we learned about in part one, Chapter 3. In Chapter 2 and 3 we discussed ways of breaking the texture sets out of a scene, but often as artists we are simply handed a list of what textures to make for a level. And often that list is incomplete. Usually a texture artist works from a checklist like the list below. Either way reading chapter 3 would benefit you here. Just working from a list and not understanding the process that it took to generate that list prevents you from contributing and helping to refine the process of texturing the world. To put some of the discussion from Part One into effect here, let’s break out the texture set for our castle level.

The castle’s base textures will consist of the following:

* Stone tiling for walls
* Stone tiles for courtyard, dirtier than walls
* Wooden tiles for the floors
* Wood, slatted, for the ceiling
* Base wood (four-way tile) for beams and stairs

The castle’s ornamental textures will consist of the following:

* Carpet
* Banner 1
* Banner 2
* Door 1 – Massive portal
* Door 2 – Smaller side door
* Grate1 - Between beams
* Grate 2 – Between holes in floor

Other elements of the texture set include:

* Skybox textures. If we had lava, water, or some other special surface we would include it here.

Some of these textures have been made in Part One of this book, and the rest we will look at in the upcoming chapters, as they are needed. In any case, you can get the complete textures from the companion CD-ROM if your interest lies in building the level and not creating the textures for it.

If this texture set request were handed to you as a texture artist, the list would contain a great deal more information. For example, the following instruction set has more information about creating a carpet for ornamental textures, but even this sample does not contain all the information a texture artist will need to create it properly in the context of a complex game:

@@@ Production: Special treatment of the following. Treat it like a code listing, but without the "Listing X.X" in the Code Cap. Thanks! --Don @@@

Sample Instructions: Ornamental Textures

```
Texture name: orn_int_crpt_001.bmp
Format: BMP
File size: 256x256, indexed
Level: Old Castle
Description: This texture is an old carpet* that lies in the old castle
entrance. The texture must have crossed chains in the pattern that prominently
appear to the gamer, but other than that, you can get creative. The chains
should not be too overstated. Keep in mind that this castle was once the
dwelling of a good king, so maybe the carpet should reflect this.
* This carpet covers a secret door.
```

As you can see, just making this small simple castle level in the context of a larger, more complex game can get demanding. In addition to the creation of the textures, we have the technology, storylines, and other aspects to pay attention to. For example, it was mentioned in the example above that this was once the castle of a good king, but now maybe it is a run down castle inhabited by evil monsters. The castle design and textures will have to reflect both the good king who once lived here, and the new monster inhabitants. The carpet for example may have the pattern and look of a once nice piece, but will now be stained and torn. The tiles on the floors will reflect this as well, having once been part of an opulent palace that is now a dark and smelly dwelling of monsters.

You may have noticed that the texture set contains some grates. These will be a few textures we will create that use transparency to help break the blocky look of the level. A grate, a chain or two, and some cobwebs will add some “delicacy” to the objects in the world. See Figure 7.9.

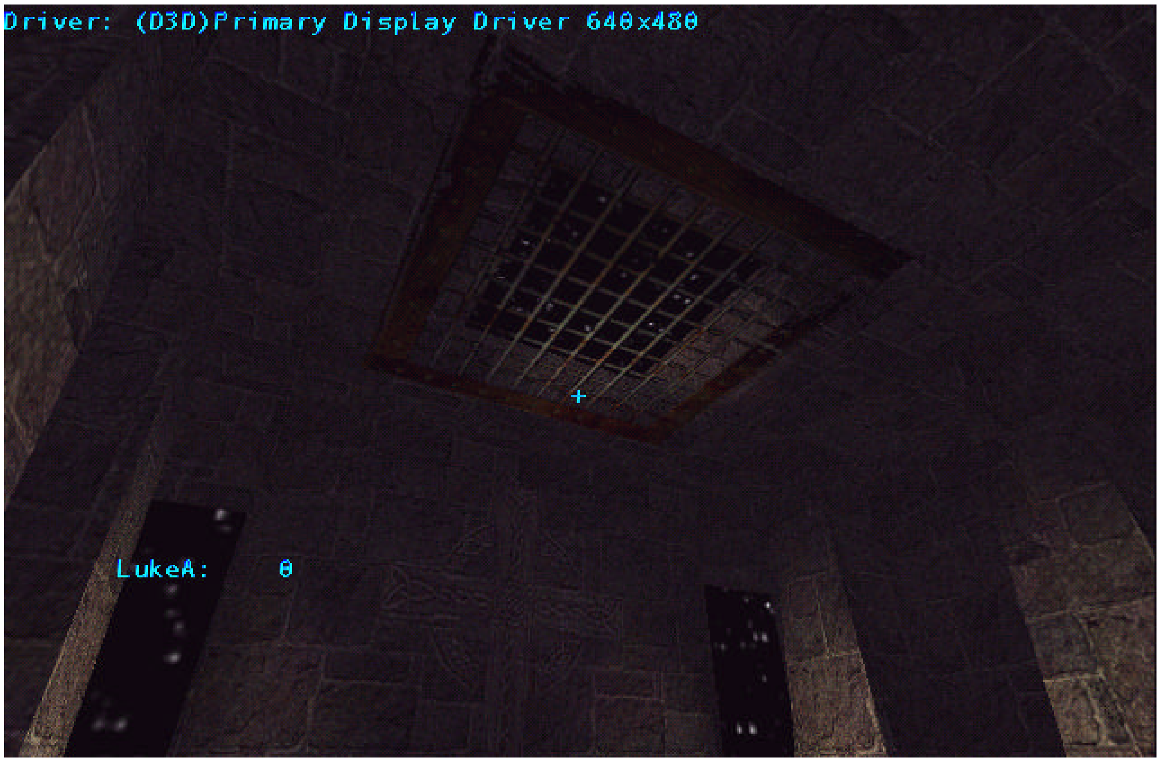


Figure 7.9 A few well-placed objects using transparency and good textures can help break the solid blocky look in a game level.

A final note: If you make your own images, save all of them in a folder where you can find them again. Remember to back up all your work and save the larger high-color versions of these images. For Genesis, you will be creating images that are 256x256 and 256 Adaptive Color. If you don't feel like creating the images, don't worry. All the images are included on the companion CD-ROM, as well as in the Texture Library format ready to use with Genesis 3D.

Moving On

In this chapter we looked at the basics of how to design and develop a level that will look and play professionally, and it really has a lot to do with planning and proper implementation. I hope you see that one theme of this book is that quite often the difference between a professional result and an amateur result is a few more moments of work, a few more touches to your work, a little more thought and planning upfront. Next chapter we will start looking at Genesis 3D tool set. Will do a quick over view of what Genesis is, how you install it on your computer to get it up and running.